Meta-mathematical aspects of Martin-Löf's type theory

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Valentini, Silvio

Meta-mathematical aspects of Martin-Löf's type theory
S. Valentini – [S.l. : s.n.]. – Ill.
Thesis Nijmegen – With ref. – With summary.
ISBN: 90-9013777-7
Subject headings: constructive type theory

Meta-mathematical aspects of Martin-Löf's type theory

een wetenschappelijke proeve op het gebied van de
Natuurwetenschappen, Wiskunde en Informatica

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Katholieke Universiteit Nijmegen,
volgens besluit van het College van Decanen
in het openbaar te verdedigen
op donderdag 22 juni 2000,
des namiddags om 1.30 uur precies

door

Silvio Luigi Maria Valentini

geboren op 6 january 1953 te Padova, Italia

Promotor: prof. dr. H.P. Barendregt

Manuscript commissie:

prof. dr. Peter Aczel (University of Manchester, UK)
prof. dr. Dirk van Dalen (Utrecht University, NL)
dr. Herman Geuvers (University of Nijmegen, NL)

Contents

Samenvatting

Vanaf de 70-er jaren heeft Martin-Löf in een aantal opeenvolgende varianten de intuitionistische typetheorie ontwikkeld. (Zie [Mar84, NPS90].) Het oorspronkelijke doel was om een formeel systeem voor constructieve wiskunde te definiëren, maar al snel werd ook het belang van de theorie voor de informatica ingezien.

In dit proefschrift geven we eerst een algemene inleiding in Martin-Löfs typetheorie. Vervolgens bespreken we een aantal van de belangrijkste meta-mathematische eigenschappen. Daarna behandelen we enkele toepassingen van de theorie binnen de constructieve wiskunde en de theoretische informatica. Tot slot analyseren we een aantal mogelijke uitbreidingen van de theorie met impredicatieve verzamelingsconstructies.

In hoofdstuk 3 worden de belangrijkste meta-mathematische eigenschappen bewezen. Dit hoofdstuk bevat een volledig bewijs van de 'canonieke vorm stelling' (*canonical form theorem*), waaruit de bekende corollaria van een normalisatie stelling volgen, zoals de consistentie van de theorie, de disjunctie en existentie eigenschappen en de totale correctheid van partieel correcte programma's. We beschouwen de theorie zoals die gepresenteerd is in [Mar84], die universa en extensionele gelijkheid bevat en waarvoor een normalisatie stelling in zijn standaard vorm niet geldt. Het bewezen resultaat in dit hoofdstuk zegt dat iedere gesloten welgetypeerde term in een aantal stappen gereduceerd kan worden naar een term in canonieke vorm.

In hoofdstuk 4 worden een aantal meta-mathematische eigenschappen geanalyseerd. We laten eerst zien dat de bekende intuitionistische karakterisering van de beslisbaarheid van predikaten equivalent is aan het bestaan van een *beslissingsfunctie*. Daarna bewijzen we een intuitionistische versie van de stelling van Cantor. Preciezer: we laten zien dat er geen surjectieve functie bestaat van de verzameling van natuurlijke getallen naar de verzameling van functies van natuurlijke getallen naar natuurlijke getallen. Tenslotte illustreren we het 'vergeet-herstel principe' *forget-restore principle* dat in [SV98] werd geïntroduceerd om uit te leggen wat abstractie in constructieve zin betekent. We doen dit door een eenvoudig voorbeeld in Martin-Löfs typetheorie te beschouwen, namelijk uitspraken van de vorm $A\ \mathsf{true}$. De betekenis van $A\ \mathsf{true}$ is dat er een element a is waarvoor $a \in A$ geldt, maar het maakt niet uit welke specifieke a het is. De overgang van de uitspraak $a \in A$ naar de uitspraak $A\ \mathsf{true}$ is een duidelijk voorbeeld van een vergeet proces. In dit hoofdstuk laten we zien dat dit een vergeet proces in constructieve zin is, daar we uit een bewijs van de uitspraak $A\ \mathsf{true}$, een element a kunnen reconstrueren waarvoor $a \in A$.

In hoofdstuk 5 laten we zien hoe een predicatieve lokale verzamelingenleer kan worden ontwikkeld binnen Martin-Löfs typetheorie. In de literatuur vindt men verschillende voorstellen voor een ontwikkeling van verzamelingenleer, binnen Martin-Löfs typetheorie of zodanig dat ze binnen Martin-Löfs typetheorie geïnterpreteerd zouden kunnen worden. Het voornaamste verschil tussen de benaderingen in de literatuur en de benadering die we hier kiezen is dat wij niet eisen dat een deelverzameling zelf weer een verzameling is. In de andere benaderingen is het doel om de bekende basis constructoren uit de typetheorie toe te passen op een algemeen verzamelingsbegrip, inclusief verzamelingen verkregen met comprehensie (over een bestaande verzameling), of om een axiomatische verzamelingenleer te definiëren waarvan de axioma's een constructieve interpretatie hebben. De theorie van deelverzamelingen die wij hier voorstellen is een soort 'typeloze' verzamelingenleer, gelokaliseerd binnen één verzameling. We bewijzen dat de gehele ontwikkeling binnen typetheorie gedaan kan worden. Uiteraard zijn niet alle axioma's van de klassieke verzamelingenleer geldig in deze benadering. In het bijzonder is het onmogelijk om de axioma's af te leiden die geen constructieve betekenis hebben, zoals de machtsverzamelingsconstructie.

Hoofdstuk 6 behandelt de ontwikkeling van een aantal niet-triviale programma's in Martin-Löfs typetheorie. Dit kan gezien worden als een studie naar abstractie in functioneel programmeren. Door de zeer krachtige type-constructies en het ingebouwde 'proposities-als-types' principe, ondersteunt Martin-Löfs typetheorie de ontwikkeling van bewijsbaar-correcte programma's. In plaats van te werken aan specifieke problemen, specificeren we in dit hoofdstuk klassen van problemen en we ontwikkelen algemene oplossingen voor deze klassen.

Tot slot is er hoofdstuk 7, waar we een uitbreiding bestuderen van Martin-Löfs intensionele typetheorie met een verzamelingsconstructor $\mathcal{P}$, zodat de elementen van $\mathcal{P}(S)$ de deelverzamelingen van S zijn. Als we voor zo'n uitbreiding een vorm van extensionaliteit op de gelijkheid van deelverzamelingen eisen (wat natuurlijk is), blijkt deze uitbreiding klassiek te zijn. Het hoofd-

stuk wordt afgesloten door te laten zien dat het voornaamste probleem met de definitie van een machtsverzamelingsconstructie hem zit in de vereiste extensionele gelijkheid tussen deelverzamelingen. Om precies te zijn: we laten zien dat niet alleen de machtsverzamelingsconstructie de logica klassiek maakt, maar dat klassieke logica al een gevolg is van de mogelijkheid om een verzameling te definiëren waarvan de elementen precies de eindige verzamelingen van een gegeven verzameling zijn. Zelfs is klassieke logica al een gevolg van de mogelijkheid om quotiëntverzamelingen te maken en tevens al van de mogelijkheid om een extensionele verzameling te definiëren wier elementen twee deelverzamelingen zijn.

De eerste appendix bevat de 'theorie van expressies' *expression theory*, een soort getypeerde λ-calculus waar een definitiemechanisme wordt gebruikt in plaats van λ-abstractie. Deze theorie is nodig om de syntax van Martin-Löfs constructieve typetheorie in uit te drukken. De tweede appendix bevat alle regels van de theorie. Deze kan door het proefschrift heen gebruikt worden als een referentie. Modulo kleine variaties zijn dit de standaard regels die op veel plaatsen in de literatuur gevonden kunnen worden. Het leek ons een goed idee om ze te bij te voegen in het proefschrift.

Curriculum vitae

Silvio Valentini was born on January 6^{th} 1953 in Padova, Italy. He lives in via Vittor Pisani n. 14, in Padova, Italy (tel. +39 049 802 44 86).

He attended the *corso di laurea* in Mathematics at the University of Padova and obtained the *laurea* in Mathematics on July 8^{th} 1977, with the thesis 'L'uso del calcolo dei predicati per la scrittura di algoritmi' (in Italian) under the direction of Professor Giovanni Sambin of the Mathematical Department of the University of Padova and Professor Enrico Pagello of L.A.D.S.E.B. of C.N.R., the National Council for Researches.

Starting October 1^{st} 1977, he won a grant of C.N.R., that he used at the Mathematical Institute of the University of Siena, headed by Professor Roberto Magari.

After November 1^{st} 1980, he became a *Ricercatore Universitario Confermato* at the Mathematical Institute of the University of Siena.

After April 2^{nd} 1984, he was a *Ricercatore Universitario Confermato* at the Mathematical Department of the University of Padova.

Then, after October 25^{th} 1987, he has become an Associate Professor in Mathematical Logic at the Computer Science Department of the University of Milan.

Now, after November 1^{st} 1991, he is an Associate Professor in Mathematical Logic at the Mathematical Department of the University of Padova.

Chapter 1

Introduction

1.1 Introduction

Since the 70s Martin-Löf has developed, in a number of successive variants, an Intuitionistic Theory of Types [Mar84, NPS90]. The initial aim was to provide a formal system for constructive mathematics but the relevance of the theory also in computer science was soon recognized. In fact, from an intuitionistic perspective, to define a constructive set theory is completely equivalent to define a logical calculus [How80] or a language for problem specification [Kol32], and hence the topic is of immediate relevance both to mathematicians, logicians and computer scientists. Moreover, since an element of a set can also be seen as a proof of the corresponding proposition or as a program which solves the corresponding problem, the intuitionistic theory of types is also a functional programming language with a very rich type structure and an integrated system to derive correct programs from their specification [PS86]. These pleasant properties of the theory have certainly contributed to the interest for it arisen in the computer science community, especially among those people who believe that program correctness is a major concern in the programming activity [BCMS89].

1.2 Outline of the thesis

In this thesis we will first give a general introduction to Martin-Löf's Type theory and then we will discuss some of its main meta-mathematical properties. Some applications of the theory both to constructive mathematics development and theoretical computer science will follow. Finally some extensions of the theory with impredicative set constructors will be analyzed.

1.2.1 General Introduction

In chapter 2 we will give an introduction to Martin-Löf's type theory by presenting the main ideas on which the theory is based and by showing some examples.

No theorem will be proved in this chapter, but we think that a general explanation of the ideas on which the theory is built on is necessary in order to have a feeling for the theory before beginning the meta-mathematical study. We think that this chapter is going to be useful to any reader which does not know Martin-Löf's type theory. In fact, only after a basic comprehension of the general approach to set construction and proposition definition will be achieved, it will be possible to understand the meaning of the technical mathematical results in the next chapters. Indeed, these results are consequences of such a basic philosophical attitude even if sometime the mathematical subtle details of their proofs can hide the intuitive content of the statements of the theorems; moreover their relevance can be undertaken if it is not clear that they are important in showing how the basic ideas are working and having effects which can be explained in mathematical terms.

For this reason in this chapter we will introduce the theory by analyzing its syntax and explaining its semantics in term of computations. Then, the general idea of inductive set will be

introduced and some simple example of program development will be shown. Finally two less tivial examples will be developed with some details: a computer memory and a Turing machine.

Most of the content of this chapter was presented in [Val96a].

1.2.2 Canonical form theorem

In chapter 3, the main meta-mathematical properties of the theory will be proved. Here, you will find the complete proof of the *canonical form theorem* that allows to obtain most of the standard consequence of a normalization theorem that is, the consistency of the theory, the disjunction and existential properties and the total correctness of any partially correct program.

In this chapter we will consider the theory presented in [Mar84] which contains both universes and extensional equality; it is well know that a standard normalization theorem does not hold for such a theory (see the introduction of the chapter for a proof of this result). On the other hand this theory is often the most useful in developing constructive mathematics and hence some kind of normal form result is useful for it.

The result that we will prove in this chapter states that any closed typed term, whose derivation has no open assumption, can be reduced by a sequence of reductions into an equivalent one in canonical form, that is, a sort of external normal form, and that the proof of any provable judgement can be transformed into an equivalent one in introductory form. These facts are sufficient to obtain most of the usual consequence of a standard normalization theorem.

It is easy to extend the proof to a theory which contains both intensional and extensional equality and it is even possible to prove a strong normalization theorem if only intensional equality is present, nevertheless we think that the theory that we presented and the technique that we used to prove the canonical form theorem are interesting enough to deserve a deep study in a case where the full normalization theorem does not hold.

The content of this chapter is mainly contained in [BV92].

1.2.3 Properties of type theory

In chapter 4 some meta-mathematical properties of the theory will be analyzed.

We will first show that the usual intuitionistic characterization of the decidability of the propositional function $B(x)$ prop $[x : A]$, that is, to require that the predicate $(\forall x \in A)\ B(x) \vee \neg B(x)$ is provable, is equivalent to require that there exists a *decision function*, that is a function $\phi : A \to \mathsf{Boole}$ such that $(\forall x \in A)\ (\phi(x) =_{\mathsf{Boole}} \mathsf{true}) \leftrightarrow B(x)$. This result turns out to be very useful in many applications of Martin-Löf's type theory and its proof is interesting since it require to use some of the peculiarities of the theory, namely the presence of universes and the fact that an intuitionistic axiom of choice is provable because of the strong elimination rule for the existential quantifier.

The results of this section can be found in [Val96].

Then, we will prove that an intutionistic version of Cantor's theorem holds. In fact, we will show that there exists no surjective function from the set of the natural numbers N into the set $\mathsf{N} \to \mathsf{N}$ of the functions from N into N. As the matter of fact a similar result can be proved for any not-empty set A such that there exists a function from A into A which has no fixed point, as is the case of the successor function for the set N.

This theorem was first presented in [MV96].

Finally, the "forget-restore" principle, introduced in [SV98] in order to explain what can be considered a constructive way to operate an abstraction, will be illustrated by analyzing a simple case in Martin-Löf's type theory. Indeed, type theory offers many ways of "forgetting" information; what will be analyzed in this section is the form of judgment A true. The meaning of A true is that there exists an element a such that $a \in A$ holds but it does not matter which particular element a is (one should compare this approach with the notion of *proof irrelevance* in [Bru80]). Thus, to pass from the judgment $a \in A$ to the judgment A true is a clear example of the forgetting process.

In this section we will show that it is a constructive way of forgetting since, provided that there is a proof of the judgment A true, an element a such that $a \in A$ can be re-constructed.

The results of this section have been presented in [Val98].

1.2.4 Subset theory

In chapter 5 we will show how a predicative local set theory can be developed within Martin-Löf's type theory.

In fact, a few years' experience in developing constructive topology in the framework of type theory has taught us that a more liberal treatment of subsets is needed than what could be achieved by remaining literally inside type theory and its traditional notation. To be able to work freely with subsets in the usual style of mathematics one must come to conceive them like any other mathematical object and have access to their usual apparatus.

Many approaches were proposed for the development of a set theory within Martin-Löf's type theory or in such a way that they can be interpreted in Martin-Löf's type theory (see for instance [NPS90], [Acz78], [Acz82], [Acz86], [Hof94], [Hof97] and [HS95]).

The main difference between our approach and these other ones stays on the fact that we do not require to a subset to be a set while in general in the other approaches the aim is to apply the usual set-constructors of basic type theory to a wider notion of set, which includes sets obtained by comprehension over a given set (see [NPS90], [Hof94], [Hof97] and [HS95]) or to define an axiomatic set theory whose axioms have a constructive interpretation (see [Acz78], [Acz82], [Acz86]); hence the justification of the validity of the rules and axioms for sets must be given anew and a new justification must be furnished each time the basic type theory or the basic axioms are modified. On the other hand the subset theory that we proposed here is a sort of type-less set theory localized to a set and we prove that all of its development can be done within type theory without losing control, that is by "forgetting" only information which can be restored at will. This is reduced to the single fact that, for any set A, the judgment $A\ \mathsf{true}$ holds if and only if there exists a such that $a \in A$, and it can be proved once and for all, see [Val98].

In this chapter we will provide with all the main definitions for a subset theory; in particular, we will state the basic notion of subset U of a set S, that is we will identify U with a propositional function over S. Hence a subset can never be a set and thus no membership relation is defined between U and the elements of S. This is the reason why our next step will be the introduction of a new membership relation between an element a of S and a subset U of S, which will turn out to hold if and only if the proposition $U(a)$ is true. Then the full theory will be developed based on these ideas, that is, the usual set-theoretic operations will be defined in terms of logical operations. Of course not all of the classical set theoretic axioms are satisfied in this approach. In particular it is not possible to validate the axioms which do not have a constructive meaning like, for instance, the power-set construction.

The topics of this section were first presented in [SV98].

1.2.5 Program development

Chapter 6 is devoted to the development of some non-trivial program within Martin-Löf's type theory and it can be considered like a sort of case study in abstraction in functional programming.

In fact, as regards computing science, through very powerful type-definitions facilities and the embedded principle of "propositions as types", Martin-Löf's type theory primarily supplies means to support the development of proved-correct programs, that is, it furnishes both the sufficient expressive power to allows general problem descriptions by means of type definitions and meanwhile a powerful deductive system where a formal proof that a type representing a problem is inhabited is ipso facto a functional program which solves such a problem. Indeed here type checking achieves its very aim, namely that of avoiding *logical* errors.

There are a lot of works (see for instance [Nor81, PS86]) stressing how it is possible to write down, within the framework of type theory, the formal specification of a problem and then develop a program meeting this specification. Actually, examples often refer to a single, well-known algorithm which is formally derived within the theory. The analogy between a mathematical constructive proof and the process of deriving a correct program is emphasized. Formality is necessary, but it is well recognized that the master-key to overcome the difficulties of formal reasoning, in mathematics as well as in computer science, is abstraction and generality. Abstraction mechanisms are very well offered by type theory by means of assumptions and dependent types.

In this chapter we want to emphasize this characteristic of the theory. Thus, instead of specifying a single problem we will specify classes of problems and we will develop general solutions for them.

The content of this section can also be found in [Val96b].

1.2.6 Forbidden constructions

To end with, there is chapter 7 where it will be analyzed an extension of Martin-Löf's intensional type theory by means of a set constructor $\mathcal{P}$ such that the elements of $\mathcal{P}(S)$ are the subsets of the set S. Since it seems natural to require some kind of extensionality on the equality among subsets, it turns out that such an extension cannot be constructive. In fact we will prove that this extension is classic, that is $(A \vee \neg A)$ true holds for any proposition A.

In the literature there are examples of intuitionistic set theories with some kind of power-set constructor. For instance, one can think to a *topos*, where a sort of "generalized set theory" is obtained by associating with any topos its internal language (see [Bel88]), or to the Calculus of Constructions by Coquand and Huet, where the power of a set S can be identified with the collection of the functions from S into prop. But in the first case problems arise because of the impredicativity of the theory and in the second because prop cannot be a set from a constructive perspective and hence also the collection of the functions from a set S into prop cannot be a set. Of course, there is no reason to expect that a second order construction becomes constructive only because it is added to a theory which is constructive. Indeed, we will prove that even the fragment iTT of Martin-Löf's type theory, which contains only the basic set constructors, i.e. no universe and no well-orders, and the *intensional* equality, cannot be extended with a power-set constructor in a way compatible with the usual explanation of the meaning of the connectives, if the power-set is the collection of all the subsets of a given set equipped with some kind of extensional equality. In fact, by using the so called intuitionistic axiom of choice, it is possible to prove that, given any power-set constructor, which satisfies the conditions that we will illustrate, classical logic is obtained. A crucial point in carrying on our proof is the uniformity of the equality condition expressing extensionality on the power-set.

The chapter is completed by showing that the main problem in the definition of the power-set constructor is the required extensionality among subsets. In fact it will be proved that not only the power-set constructor allows to obtain classical logic but that it is a consequence also of the possibility to define a set whose elements are the finite subsets of a given set, of the possibility to define quotient sets and even of the possibility to define an extensional set whose elements are two subsets.

The proofs in this chapter were first presented in [MV99] and [Val00].

1.2.7 Appendices

Finally, two appendices follow. The first contains *expression theory*, that is, a sort of typed lambda-calculus where abbreviating definitions are used instead that λ-abstraction, which is necessary to have a formal theory where the syntax for Martin-Löf's constructive type theory can be expressed.

Expression theory was first presented in [BV89].

The second appendix contains all of the rules of the theory and it can be used like a reference along all the thesis. With some small variants, such rules can be found in many printed papers (see for instance [Mar84, NPS90, BV92] and others), but we thought that it could be handy to have them within the thesis.

Chapter 2

Introduction to Martin-Löf's type theory

2.1 Introduction

Since the 70s Martin-Löf has developed, in a number of successive variants, an Intuitionistic Theory of Types [Mar84, NPS90] (ITT for short in the following). The initial aim was to provide a formal system for constructive mathematics but the relevance of the theory also in computer science was soon recognized. In fact, from an intuitionistic perspective, to define a constructive set theory is completely equivalent to define a logical calculus [How80] or a language for problem specification [Kol32], and hence the topic is of immediate relevance both to mathematicians, logicians and computer scientists. Moreover, since an element of a set can also be seen as a proof of the corresponding proposition or as a program which solves the corresponding problem, ITT is also a functional programming language with a very rich type structure and an integrated system to derive correct programs from their specification [PS86]. These pleasant properties of the theory have certainly contributed to the interest for it arisen in the computer science community, especially among those people who believe that program correctness is a major concern in the programming activity [BCMS89].

To develop ITT one has to pass through four steps:

- The first step is the definition of a theory of *expressions* which both allows to abstract on variables and has a decidable equality theory; indeed the first requirement is inevitable to gain the needed expressive power while the last is essential in order to guarantee a mechanical verification of the correct application of the rules used to present ITT. Here the natural candidate is a sort of simple typed lambda calculus which can be found in the appendix A.

- The second step is the definition of the *types* (respectively *sets*, *propositions* or *problems*) one is interested in; here the difference between the classical and the intuitionistic approach is essential: in fact a proposition is not merely an expression supplied with a truth value but rather an expression such that one knows what counts as one of its *verifications* (respectively one of its *elements* or one of the *programs* which solves the problem).

- The third step is the choice of the *judgments* one can express on the types introduced in the previous step. Four forms of judgment are considered in ITT:

  1. the first form of judgment is obviously the judgment which asserts that an expression is a type;
  2. the second form of judgment states that two types are equal;
  3. the third form of judgment asserts that an expression is an element of a type;
  4. the fourth form of judgment states that two elements of a type are equal.

- The fourth step is the definition of the computation rules which allow to execute the programs defined in the previous point.

In this chapter we will show some of the standard sets and propositions in [Mar84] and some examples of application of the theory to actual cases, while for the study of the main meta-mathematical results on ITT the reader is asked to wait for the next chapters.

2.2 Judgments and Propositions

In a classical approach to the development of a formal theory one usually takes care to define a formal language only to specify the syntax (s)he wants to use while as regard to the intended semantics no *a priori* tie on the used language is required. Here the situation is quite different; in fact we want to develop a *constructive* set theory and hence we can assume no *external* knowledge on sets; then we do not merely develop a suitable syntax to describe something which exists *somewhere*. Hence we have "to put our cards on the table" at the very beginning of the game and to declare the kind of judgments on sets we are going to express. Let us consider the following example: let A be a set,then

$$a \in A$$

which means that a is an element of A, is one of the form of judgment we are interested in (but also the previous "A is a set" is already a form of judgment!). It is important to note that a logical calculus is meaningful only if it allows to derive judgments. Hence one should try to define it only after the choice of the judgments (s)he is interested in. A completely different problem is the definition of the suitable notation to express such a logical calculus; in this case it is probably more correct to speak of a *well-writing* theory instead of a logical calculus (see appendix A to see a well-writing theory suitable for ITT).

Let us show the form of the judgments that we are going to use to develop our constructive set theory. The first is

$$\text{(type-ness)} \quad A \ \mathsf{type}$$

(equivalently A set, A prop and A prob) which reads "A is a type" (respectively "A is a set", "A is a proposition" and "A is a problem") and states that A is a type. The second form of judgment is

$$\text{(equal types)} \quad A = B$$

which, provided that A and B are types, states that they are equal types. The next judgment is

$$\text{(membership)} \quad a \in A$$

which states that a is an element of the type A. Finally

$$\text{(equal elements)} \quad a = b \in A$$

which, provided that a and b are elements of the type A, states that a and b are equal elements of the type A.

2.3 Different readings of the judgments

Here we can see a peculiar aspect of a constructive set theory: we can give many different readings for the same judgment. Let us show some of them

$\mathbf{A}$ type	$\mathbf{a} \in \mathbf{A}$
A is a set	a is an element of A
A is a proposition	a is a proof of A
A is a problem	a is a method which solves A

The first reading conforms to the definition of a constructive set theory. The second one, which links type theory with intuitionistic logic, is due to Heyting [Hey56, How80]: it is based on the identification of a proposition with the set of its proofs. Finally the third reading, due to Kolmogorov [Kol32], consists on the identification of a problem with the set of its solutions.

2.3.1 On the judgment A set

Let us explain the meaning of the various forms of judgment; to this aim we may use many different philosophical approaches: here it is convenient to commit to an epistemological one.

What is a set? A set is defined by prescribing how its elements are formed.

Let us work out an example: the set N of natural numbers. We state that natural numbers form a set since we know how to construct its elements.

$$0 \in \mathsf{N} \qquad \frac{a \in \mathsf{N}}{\mathsf{s}(a) \in \mathsf{N}}$$

i.e. 0 is a natural number and the successor of any natural number is a natural number.

Of course in this way we only construct the *typical* elements, we will call them the *canonical elements*, of the set of the natural numbers and we say nothing on elements like $3 + 2$. We can recognize also this element as a natural number if we understand that the operation $+$ is just a method such that, given two natural numbers, provides us, by means of a calculation, with a natural number in canonical form, i.e. $3+2 = \mathsf{s}(2+2)$; this is the reason why, besides the judgment on membership, we also need the equal elements judgment. For the type of the natural numbers we put

$$0 = 0 \in \mathsf{N} \qquad \frac{a = b \in \mathsf{N}}{\mathsf{s}(a) = \mathsf{s}(b) \in \mathsf{N}}$$

In order to make clear the meaning of the judgment A set let us consider another example. Suppose that A and B are sets, then we can construct the type $A \times B$, which corresponds to the cartesian product of the sets A and B, since we know what are its canonical elements:

$$\frac{a \in A \quad b \in B}{\langle a, b\rangle \in A \times B} \qquad \frac{a = c \in A \quad b = d \in B}{\langle a, b\rangle = \langle c, d\rangle \in A \times B}$$

So we explained the meaning of the judgment A set but meanwhile we also explained the meaning of two other forms of judgment.

What is an element of a set? An element of a set A is a method which, when executed, yields a canonical element of A as result.

When are two elements equal? Two elements a, b of a set A are equal if, when executed, they yield equal canonical elements of A as results.

It is interesting to note that one cannot construct a set if (s)he does not know how to produce its elements: for instance the subset of the natural numbers whose elements are the code numbers of the recursive functions which do not halt when applied to 0 is *not* a type in ITT, due to the halting problem. Of course it *is* a subset, since there is a way to describe it, and a suitable subset theory is usually sufficient in order to develop a great deal of standard mathematics (see chapter 5 or [SV98]).

2.3.2 On the judgment A prop

We can now immediately explain the meaning of the second way of reading the judgment A type, i.e. to answer to the question: *What is a proposition?*

Since we want to identify a proposition with the set of its proofs, in order to answer to this question we have only to repeat for proposition what we said for sets: a proposition is defined by laying down what counts as a proof of the proposition.

This approach is completely different from the classical one; in the classical case a proposition is an expression provided with a truth value, while here to state that an expression is a proposition one has to clearly state what (s)he is willing to accept as one of its proofs. Consider for instance the proposition $A\&B$: supposing A and B are propositions, then $A\&B$ is a proposition since we can state what is one of its proofs: a proof of $A\&B$ consists of a proof of A together with a proof of B, and so we can state

$$\frac{a \in A \quad b \in B}{\langle a, b\rangle \in A\&B}$$

but then $A\&B \equiv A \times B$ and, more generally, we can identify sets and propositions.

A lot of propositions

Since we identify sets and propositions, we can construct a lot of new sets if we know how to construct new propositions, i.e. if we can explain what is one of their proofs. Even if their intended meaning is completely different with respect to the classical case, we can recognize the following propositions.

a proof of	**consists of**
$A\&B$	$\langle a, b\rangle$, where a is a proof of A and b is a proof of B
$A \vee B$	$\mathsf{inl}(a)$, where a is a proof of A or $\mathsf{inr}(b)$, where b is a proof of B
$A \to B$	$\lambda(b)$, where b(x) is a proof of B provided that x is a proof of A
$(\forall x \in A)\ B(x)$	$\lambda(b)$, where b(x) is a proof of $B(x)$ provided that x is an element of A
$(\exists x \in A)\ B(x)$	$\langle a, b\rangle$, where a is an element of A and b is a proof of $B(a)$
$\bot$	nothing

It is worth noting that the intuitionistic meaning of the logical connectives allows to recognize that the connective $\to$ is just a special case of the quantifier $\forall$, provided the proposition $B(x)$ does not depend on the variable x, and the connective $\&$ is a special case of the quantifier $\exists$ under the same assumption.

Let us see which sets correspond to the propositions we have defined so far.

The proposition	**corresponds to the set**
$A\&B$	$A \times B$, the cartesian product of the sets A and B
$A \vee B$	$A + B$, the disjoint union of the sets A and B
$A \to B$	$A \to B$, the set of the functions from A to B
$(\forall x \in A)\ B(x)$	$\Pi(A, B)$, the cartesian product of a family $B(x)$ of types indexed on the type A
$(\exists x \in A)\ B(x)$	$\Sigma(A, B)$, the disjoint union of a family $B(x)$ of types indexed on the type A
$\bot$	$\emptyset$, the empty set

2.4 Hypothetical judgments

So far we have explained the basic ideas of ITT; now we want to introduce a formal system. To this aim we must introduce the notion of hypothetical judgment: a hypothetical judgment is a judgment expressed under assumptions. Let us explain what is an assumption; here we only give some basic ideas while a formal approach can be found in the next chapter 3 or in [BV92]. Let A be a type; then the assumption

$$x : A$$

means both:

1. a variable declaration: the variable x has type A
2. a logical assumption: x is a hypothetical proof of A.

The previous is just the simplest form of assumption, but we can also use

$$y : (x : A)\ B$$

which means that y is a function which maps an element $a \in A$ into the element $y(a) \in B$, and so on, by using assumptions of arbitrary complexity (see appendix A to find some explanation on the notation we use).

Let us come back to hypothetical judgments. We start with the simplest example of hypothetical judgment; suppose A type then we can state that B is a propositional function on the elements of A by putting

$$B(x) \text{ prop } [x : A]$$

provided that we know what it counts as a proof of $B(a)$ for each element $a \in A$. For instance, one could consider the hypothetical judgment

$$x \neq 0 \to (\frac{3}{x} * x = 3) \text{ prop } [x : \mathsf{N}]$$

whose meaning is obvious.

Of course, we also have:

$$B(x) = D(x) \ [x : A]$$
$$b(x) \in B(x) \ [x : A]$$
$$b(x) = d(x) \in B(x) \ [x : A]$$

The previous are just the simplest forms of hypothetical judgments and in general, supposing

$$A_1 \text{ type}, A_2(x_1) \text{ type } [x_1 : A_1], \ldots, A_n(x_1, \ldots, x_{n-1}) \text{ type } [x_1 : A_1, \ldots, x_{n-1} : A_{n-1}]$$

we can introduce the hypothetical judgment

$$J \ [x_1 : A_1, \ldots, x_n : A_n(x_1, \ldots, x_{n-1})]$$

where J is one of the four forms of judgment that we have considered that can contain the variables $x_1, \ldots, x_n$.

2.5 The logic of types

We can now describe the full set of rules needed to describe one type. We will consider four kinds of rules: the first rule states the conditions required in order to *form* the type, the second one *introduces* the canonical elements of the type, the third rule explains how to use, and hence *eliminate*, the elements of the type and the last one how to *compute* using the elements of the type. For each kind of rules we will first give an abstract explanation and then we will show the actual rules for the cartesian product of two types.

Formation. How to form a new type (possibly by using some previously defined types) and when two types constructed in this way are equal.

Example:

$$\frac{A \text{ type} \quad B \text{ type}}{A \times B \text{ type}} \qquad \frac{A = C \quad B = D}{A \times B = C \times D}$$

which state that the cartesian product of two types is a type.

Introduction. What are the canonical elements of the type and when two canonical elements are equal.

Example:

$$\frac{a \in A \quad b \in B}{\langle a, b\rangle \in A \times B} \qquad \frac{a = c \in A \quad b = d \in B}{\langle a, b\rangle = \langle c, d\rangle \in A \times B}$$

which state that the canonical elements of the cartesian product $A \times B$ are the couples whose first element is in A and second element is in B.

Elimination. How to define functions on the elements of the type defined by the introduction rules.

Example:

$$\frac{c \in A \times B \quad d(x, y) \in C(\langle x, y\rangle) \ [x : A, y : B]}{\mathsf{E}(c, d) \in C(c)}$$

which states that to define a function on *all* the elements of the type $A \times B$ it is sufficient to explain how it works on the canonical elements.

Equality. How to compute the function defined by the elimination rule.

Example:

$$\frac{a \in A \quad b \in B \quad d(x,y) \in C(\langle x,y\rangle)\ [x:A, y:B]}{\mathsf{E}(\langle a,b\rangle, d) = d(a,b) \in C(\langle a,b\rangle)}$$

which states that to evaluate the function $\mathsf{E}(c,d)$, defined by the elimination rule, one has first to evaluate c in order to obtain a canonical element $\langle a,b\rangle \in A \times B$ and then to use the method $d : (x:A)(y:B)\ C(\langle x,y\rangle)$, provided by the second premise of the elimination rule, to obtain the value $d(a,b) \in C(\langle a,b\rangle)$.

The same approach can be used to obtain the rules for a type which is not a *logical* proposition. Let us consider the case of the type N. It is interesting to note that there is no need to change *anything* with respect to the general pattern in order to recover all the usual properties of natural numbers.

Formation:

$$\mathsf{N}\ \mathsf{set}$$

Introduction:

$$0 \in \mathsf{N} \qquad \frac{a \in \mathsf{N}}{\mathsf{s}(a) \in \mathsf{N}}$$

Elimination:

$$\frac{c \in \mathsf{N} \quad d \in C(0) \quad e(x,y) \in C(\mathsf{s}(x))[x:\mathsf{N}, y:C(x)]}{\mathsf{N}_{rec}(c,d,e) \in C(c)}$$

Equality:

$$\frac{d \in C(0) \quad e(x,y) \in C(\mathsf{s}(x))\ [x:\mathsf{N}, y:C(x)]}{\mathsf{N}_{rec}(0,d,e) = d \in C(0)}$$

$$\frac{a \in \mathsf{N} \quad d \in C(0) \quad e(x,y) \in C(\mathsf{s}(x))[x:\mathsf{N}, y:C(x)]}{\mathsf{N}_{rec}(\mathsf{s}(a),d,e) = e(a, \mathsf{N}_{rec}(a,d,e)) \in C(\mathsf{s}(a))}$$

As you see the elimination rule is an old friend: we have re-discovered the induction principle on natural numbers.

We can now consider a new kind of rules which makes explicit the computation process which is only implicit in the equality rule.

Computation:

$$\frac{c \Rightarrow 0 \quad d \Rightarrow g}{\mathsf{N}_{rec}(c,d,e) \Rightarrow g} \qquad \frac{c \Rightarrow \mathsf{s}(a) \quad e(a, \mathsf{N}_{rec}(a,d,e)) \Rightarrow g}{\mathsf{N}_{rec}(c,d,e) \Rightarrow g}$$

2.6 Some programs

Now we can develop some simple programs on natural numbers and look at their execution.

2.6.1 The sum of natural numbers

Let $x, y \in \mathsf{N}$, then the sum of x and y is defined by means of the following deduction:

$$\frac{x \in \mathsf{N} \quad y \in \mathsf{N} \quad \dfrac{[v:\mathsf{N}]_1}{\mathsf{s}(v) \in \mathsf{N}}}{x + y \equiv \mathsf{N}_{rec}(x, y, (u,v)\ \mathsf{s}(v)) \in \mathsf{N}}\ 1$$

For instance we can evaluate $3 + 2$ as follows:

$$\frac{3 \Rightarrow \mathsf{s}(2) \quad \mathsf{s}(\mathsf{N}_{rec}(2, 2, (u,v)\ \mathsf{s}(v)) \Rightarrow \mathsf{s}(2+2)}{3 + 2 \equiv \mathsf{N}_{rec}(3, 2, (u,v)\ \mathsf{s}(v)) \Rightarrow \mathsf{s}(2+2)}$$

This simple example can already suggest that ITT is a functional programming language with a strong typing system.

2.6.2 The product of natural numbers

For any $x, y \in \mathsf{N}$, we define the product of x and y by means of the following deduction which makes use of the definition of the sum of the previous section.

$$\dfrac{\begin{array}{ccc} & & y : \mathsf{N} \quad [v : \mathsf{N}]_1 \\ & & \vdots \\ x \in \mathsf{N} & 0 \in \mathsf{N} & y + v \in \mathsf{N} \end{array}}{x * y \equiv \mathsf{N}_{rec}(x, 0, (u, v)\ y + v) \in \mathsf{N}}\ 1$$

In general, the recursive equation with unknown f:

$$\begin{cases} f(0) = k \in A \\ f(\mathsf{s}(x)) = g(x, f(x)) \in A \end{cases}$$

is solved in ITT by

$$f(x) \equiv \mathsf{N}_{rec}(x, k, (u, v)\ g(u, v))$$

and it is possible to prove that

$$\mathsf{N}_{rec}(x, k, (u, v)\ g(u, v)) \in A\ [x : \mathsf{N}, k : A, g : (u : \mathsf{N})(v : A)\ A]$$

2.7 All types are similar

Looking at the rules for the types that we have introduced till now it is possible to realize that they always follow the same pattern. First the *formation* rules state how to form the new type. For instance in order to define the type $A \to B$ we put:

$$\dfrac{A\ \mathsf{type} \quad B\ \mathsf{type}}{A \to B\ \mathsf{type}} \qquad \dfrac{A = C \quad B = D}{A \to B = C \to D}$$

The second step is the definition of the canonical elements of the type; this is the step which completely determines the type we are constructing since all the other rules directly depend on these ones. For instance, for the type $A \to B$ we state that the canonical elements are the functions $\lambda(b)$ such that $b(x) \in B\ [x : A]$.

$$\dfrac{b(x) \in B\ [x : A]}{\lambda(b) \in A \to B} \qquad \dfrac{b(x) = d(x) \in B\ [x : A]}{\lambda(b) = \lambda(d) \in A \to B}$$

The elimination rule is now determined; it states that the *only* elements of the type are those introduced by the introduction rule(s) and hence that one can define a function on all the elements of the type if (s)he knows how this function works on the canonical elements. Again let us use the type $A \to B$ as a paradigmatic example; for this type we have only one introduction rule and hence in the elimination rule, besides the premise $c \in A \to B$, we have to consider only another premise, i.e. $d(y) \in C(\lambda(y))\ [y : (x : A)\ B]$ which shows how to obtain a proof of $C(\lambda(y))$ starting from a generic assumptions for the introduction rule. Now $c \in A \to B$ must be equal to $\lambda(b)$ for some $b(x) \in B\ [x : A]$ and hence by using c one has to be able to construct anything (s)he can construct starting from $b : (x : A)\ B$. Since the second premise states that we can obtain a proof $d(b)$ of $C(\lambda(b))$, then the elimination rule states that we have to be able to obtain a proof of $C(c)$, which we call $\mathsf{F}(c, d)$, depending on the two assumptions $c \in A \to B$ and $d : (y : (x : A)\ B)\ C(\lambda(y))$.

$$\dfrac{c \in A \to B \quad d(y) \in C(\lambda(y))\ [y : (x : A)\ B]}{\mathsf{F}(c, d) \in C(c)}$$

Let us now consider the equality rule: it shows how to compute the function defined by the elimination rule. Since in the elimination rule we have considered a different premise in correspondence with each introduction rule, we will need also a particular equality rule in correspondence with each introduction rule. In fact, consider the evaluation process of a function obtained by an elimination

rule for the type A. First the element of the type A which appears in the leftmost premise is evaluated into a canonical element of A in correspondence to a suitable introduction rule; then the premise(s) of this introduction rule can be substituted for the assumption(s) of the corresponding premise of the elimination rule. Let us consider the case of the type $A \to B$: we have only one introduction rule and hence we have to define one equality rule which explains how to evaluate a function $\mathsf{F}(z,d) \in C(z)\ [z : A \to B]$ when it is used on the canonical element $\lambda(b)$; the second premise of the elimination rule states that we obtain a proof $d(b)$ of $C(\lambda(b))$ simply by substituting b for y and hence we put $\mathsf{F}(\lambda(b), d) = d(b)$.

$$\frac{b(x) \in B\ [x : A] \quad d(y) \in C(\lambda(y))\ [y : (x : A)\ B]}{\mathsf{F}(\lambda(b), d) = d(b) \in C(\lambda(b))}$$

To show now a situation which is a little different, let us analyze the elimination and the equality rules for the type N. In this case we have two introduction rules and hence in the elimination rule, besides the premise $c \in \mathsf{N}$, there will be two other premises. The first, in correspondence with the introduction rule $0 \in \mathsf{N}$ which has no premise, has no assumption and hence, supposing we want to prove an instance of the propositional function $C(z)\ \mathsf{prop}\ [z : \mathsf{N}]$, it must be a proof d of the proposition $C(0)$. The second premise is more complex here than in the previous case since the second introduction rule for the type N is inductive, i.e. the premise contains the type N itself. In this case we can suppose to have proved the property $C(x)$ for the natural number x before we construct the natural number $\mathsf{s}(x)$ and hence in the elimination rule, when we are going to prove $C(\mathsf{s}(x))$, besides the assumption due to the premise of the introduction rule, we can also assume to have a proof of $C(x)$.

Since we have two introduction rules we also have two equality rules. The first concerns $\mathsf{N}_{rec}(c, d, e)$, in correspondence with the first introduction rule, i.e. when the value of c is 0. In this case the second assumption shows that $\mathsf{N}_{rec}(0, d, e) = d$. The second equality rule concerns the case the value of c is $\mathsf{s}(a)$ for some natural number a: in this case we can suppose to know that $\mathsf{N}_{rec}(a, d, e)$ is an element of $C(a)$ and hence the third assumption shows that $\mathsf{N}_{rec}(\mathsf{s}(a), d, e) = e(a, \mathsf{N}_{rec}(a, d, e))$.

We can now play a little game and see how these formal considerations work even if one has no a priori comprehension of the type (s)he is defining. Let us suppose to have the following formation and introduction rules:

Formation:

$$\frac{A\ \mathsf{type} \quad B(x)\ \mathsf{type}\ [x : A]}{\mathsf{W}(A, B)\ \mathsf{type}}$$

Introduction:

$$* \in \mathsf{W}(A, B) \qquad \frac{a \in A \quad b(y) \in \mathsf{W}(A, B)\ [y : B(a)]}{\circ(a, b) \in \mathsf{W}(A, B)}$$

Can you find the correct elimination and equality rules?

The elimination rule is completely determined by the introduction rules: there are two introduction rules and hence, besides the premise $c \in \mathsf{W}(A, B)$, there would be two minor premises. The first will have no assumption because the first introduction rule has no premise, while the second premise will have an assumption in correspondence with any premise of the second introduction rule plus an inductive assumption since this introduction rule is inductive.

Elimination:

$$\frac{\begin{array}{c}[x : A, w : (y : B(x))\mathsf{W}(A, B), z : (y : B(x))C(w(y))]_1 \\ \vdots \\ c \in \mathsf{W}(A, B) \quad d \in C(*) \quad e(x, w, z) \in C(\circ(x, w))\end{array}}{\square(c, d, e) \in C(c)}\ 1$$

Finally we have to define two equality rules in correspondence with the two introduction rules.

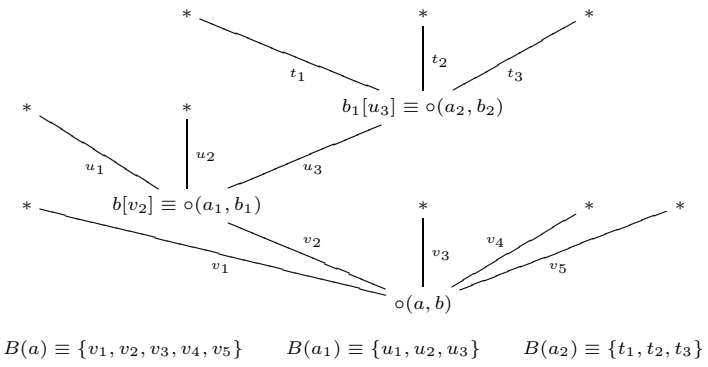


Figure 2.1: the tree $\circ(a, b)$

Equality:

$$\frac{\begin{array}{c}[x : A, w : (y : B(x))\mathsf{W}(A, B), z : (y : B(x))C(w(y))]_1 \\ \vdots \\ d \in C(*) \qquad e(x, w, z) \in C(\circ(x, w))\end{array}}{\Box(*, d, e) = d \in C(*)}\ 1$$

$$\frac{a \in A \quad b(y) \in \mathsf{W}(A, B)[y : B(a)] \quad d \in C(*) \quad e(x, w, z) \in C(\circ(x, w))}{\Box(\circ(a, b), d, e) = e(a, b, (y)\ \Box(b(y), d, e)) \in C(\circ(a, b))}\ 1$$

Can you guess what this type is? It is the type of the labeled tree: the element $* \in \mathsf{W}(A, B)$ is a leaf and $\circ(a, b) \in \mathsf{W}(A, B)$ is a node which has the label $a \in A$ and a branch which arrives at the tree $b(y) \in \mathsf{W}(A, B)$ in correspondence with each element $y \in B(a)$ (see figure 2.1).

2.8 Some applications

To understand the expressive power of ITT as a programming language, let us show some simple programs: the first is the description of a computer memory and the second one is a Turing machine.

2.8.1 A computer memory

To describe a computer memory it is convenient to introduce the type Boole of the boolean values.

Formation:

$$\mathsf{Boole}\ \mathsf{set}$$

Introduction:

$$\top \in \mathsf{Boole} \qquad \bot \in \mathsf{Boole}$$

Elimination:

$$\frac{c \in \mathsf{Boole} \quad d \in C(\top) \quad e \in C(\bot)}{\text{if } c \text{ then } d \text{ else } e \text{ endif} \in C(c)}$$

Equality:

$$\frac{d \in C(\top) \quad e \in C(\bot)}{\text{if } \top \text{ then } d \text{ else } e \text{ endif} = d \in C(\top)} \qquad \frac{d \in C(\top) \quad e \in C(\bot)}{\text{if } \bot \text{ then } d \text{ else } e \text{ endif} = e \in C(\bot)}$$

Then we obtain the type Value of a m-bits memory word by using m times the cartesian product of Boole with itself.

$$\mathsf{Value} \equiv \mathsf{Boole}^m \equiv \underbrace{\mathsf{Boole} \times \ldots \times \mathsf{Boole}}_{m}$$

By using the same construction we can define also the address space by putting

$$\mathsf{Address} \equiv \mathsf{Boole}^n$$

if we want to specify a computer with an n-bits address bus.

The definition of a computer memory is now straightforward: a memory is a map which returns a value in correspondence with any address.

$$mem \in \mathsf{Address} \to \mathsf{Value}$$

The first operation one have to define on a computer memory is its initialization at startup; the following position is one of the possible solution of the problem of memory initialization:

$$mem \equiv \lambda((x)\ \langle \bot, \ldots, \bot \rangle)$$

where $\langle a_1, \ldots, a_n \rangle$ is the obvious generalization of the operation of couple formation that we used for the cartesian product.

We can now define the two standard operations on a computer memory: reading and writing. Suppose $add \in \mathsf{Address}$ and $mem \in \mathsf{Address} \to \mathsf{Value}$, then the function $\mathsf{read}(mem, add)$, which returns the value at the memory location add of the memory mem, consists simply in applying the function mem to the address add:

$$\mathsf{read}(mem, add) \equiv mem[add] \equiv \mathsf{F}(mem, (y)\ y(add))$$

As regard to the function $\mathsf{write}(mem, add, value)$, which returns the new memory configuration after writing the value val at the location add of the memory mem, we put:

$$\mathsf{write}(mem, add, val) \equiv \lambda((x)\ \text{if } x = add \text{ then } val \text{ else } \mathsf{read}(mem, x) \text{ endif})$$

It is obvious that the computer memory we proposed here is not very realistic, but we can improve it a bit if we introduce the type ExtBoole.

Formation:

$$\mathsf{ExtBoole\ set}$$

Introduction:

$$\top \in \mathsf{ExtBoole} \quad \bot \in \mathsf{ExtBoole} \quad \omega \in \mathsf{ExtBoole}$$

where the new canonical element $\omega \in \mathsf{ExtBoole}$ can be used to specify the presence of a value still undefined in the memory.

Elimination:

$$\frac{c \in \mathsf{ExtBoole} \quad d_1 \in C(\top) \quad d_2 \in C(\bot) \quad d_3 \in C(\omega)}{\text{case } c \text{ of } \top : d_1;\ \bot : d_2;\ \omega : d_3 \text{ endcase} \in C(c)}$$

Equality:

$$\frac{d_1 \in C(\top) \quad d_2 \in C(\bot) \quad d_3 \in C(\omega)}{\text{case } \top \text{ of } \top : d_1;\ \bot : d_2;\ \omega : d_3 \text{ endcase} = d_1 \in C(\top)}$$

$$\frac{d_1 \in C(\top) \quad d_2 \in C(\bot) \quad d_3 \in C(\omega)}{\text{case } \bot \text{ of } \top : d_1;\ \bot : d_2;\ \omega : d_3 \text{ endcase} = d_2 \in C(\bot)}$$

$$\frac{d_1 \in C(\top) \quad d_2 \in C(\bot) \quad d_3 \in C(\omega)}{\text{case } \omega \text{ of } \top : d_1;\ \bot : d_2;\ \omega : d_3 \text{ endcase} = d_3 \in C(\omega)}$$

In this way we can improve the initialization function by putting

$$mem \equiv \lambda((x)\ \langle \omega, \ldots, \omega \rangle)$$

which states that at startup the state of the memory is completely unknown. In this way it becomes clear that we must write a *real* value at a particular address before we can read something meaningful at that address.

2.8.2 A Turing machine on a two symbols alphabet

Our second example is the definition of a Turing machine which uses a two symbols alphabet. [1]

Let us begin with some definitions which makes clearer what follows. The first definition concerns the movements of the head.

$$\begin{aligned} \mathsf{Move} &\equiv \mathsf{Boole} \\ \mathsf{left} &\equiv \top \\ \mathsf{right} &\equiv \bot \end{aligned}$$

Then, as regard the alphabet we put

$$\begin{aligned} \mathsf{Symb} &\equiv \mathsf{Boole} \\ 0 &\equiv \top \\ 1 &\equiv \bot \end{aligned}$$

Finally we define our Turing machine. A Turing machine is made of a *tape* where a *head* can read and write under the control of a program whose instructions are scanned by a program counter which indicates the execution *state*. Thus we can formally identify a Turing machine with a quadruple

$$\mathsf{TuringMachine} \equiv \langle prog, tape, head, state \rangle$$

where, supposing

$$\begin{aligned} \mathsf{TapeType} &\equiv \mathsf{N} \to \mathsf{Symb} \\ \mathsf{HeadPos} &\equiv \mathsf{N} \\ \mathsf{States} &\equiv \mathsf{N} \\ \mathsf{ProgType} &\equiv (\mathsf{States} \times \mathsf{Symb}) \to (\mathsf{Symb} \times \mathsf{Move} \times \mathsf{States}) \end{aligned}$$

we have

$$tape \in \mathsf{TapeType}$$

i.e. the tape is a function which gives the symbol at the n-th location of the tape if applied to the natural number n; [2]

$$head \in \mathsf{HeadPos}$$

i.e. *head* is the current position of the head over the tape;

$$state \in \mathsf{States}$$

i.e. *state* is the current execution state of the machine, and finally

$$prog \in \mathsf{ProgType}$$

i.e. *prog* is the transition function of the machine which, given the actual state and the symbol at the current position of the head over the tape, returns the symbol to write at the current position, the next movement of the head and the new state of the machine.

We can now describe the execution process of the Turing machine. Of course we have to start by considering the initial conditions, i.e. the initial configuration of the tape, the initial position of the head and the initial state of the machine. For instance we put

$$\begin{aligned} initTape &\equiv \lambda x.\ \text{if } x = 0 \text{ then } 1 \\ &\qquad \text{else if } x = 1 \text{ then } 1 \\ &\qquad\quad \vdots \\ &\qquad \text{else if } x = n \text{ then } 1 \\ &\qquad \text{else } 0 \text{ endifs} \in \mathsf{TapeType} \\ initHead &\equiv 0 \in \mathsf{HeadPos} \\ initState &\equiv 0 \in \mathsf{States} \end{aligned}$$

[1] It is well known that the use of such a simple alphabet does not limit the expressive power of the machine; here we prefer to limit the alphabet to avoid to introduce new types.

[2] In order to use the type of the natural numbers which we have already defined, we consider here Turing machines whose tape is non-finite only at the right instead of the more standard machines whose tape has both sides not finite; it is well-known that this requirement does not change the class of the computable functions.

to state that the machine works on the tape $11\ldots1000\ldots$, which contains the symbol 1 in its first n locations and 0 in all the other ones, and starts in the execution state 0 with its head over the location 0.

Let us now show how a computation step can be described. Let *prog* be a program and suppose to have an actual description of the machine determined by the actual tape, the actual position of the head and the actual state; then we want to obtain the new configuration of the tape, the new position of the head and the new state of the machine which result after the execution of one computation step. Hence we need a function

$$\mathsf{execStep} \in \mathsf{ProgType} \to (\mathsf{TapeType} \times \mathsf{HeadPos} \times \mathsf{States} \to \mathsf{TapeType} \times \mathsf{HeadPos} \times \mathsf{States})$$

such that

$$\mathsf{execStep}[prog][\langle inTape, inHead, inState\rangle] = \langle outTape, outHead, outState\rangle$$

To define $\mathsf{execStep}$ it is convenient to put $\mathsf{fst}(c) \equiv \mathsf{E}(c, (x,y)\ x)$ and $\mathsf{snd}(c) \equiv \mathsf{E}(c, (x,y)\ y)$ so that $\mathsf{fst}(\langle a,b\rangle) = a$ and $\mathsf{snd}(\langle a,b\rangle) = b$ and hence $\Pi^3_1(x) \equiv \mathsf{fst}(\mathsf{fst}(x))$, $\Pi^3_2(x) \equiv \mathsf{snd}(\mathsf{fst}(x))$ and $\Pi^3_3(x) \equiv \mathsf{snd}(x)$.

In fact we can now define $outTape$, $outHead$, $outState$ as a function of $inTape$, $inHead$ and $inState$, besides the program $prog$, as follows:

$$\begin{array}{rl} outTape & \equiv \lambda x.\ \text{if } x = inHead \\ & \quad \text{then } \Pi^3_1(prog[inState, inTape[inHead]]) \\ & \quad \text{else } inTape[x] \text{ endif} \\ outHead & \equiv \text{if } \Pi^3_2(prog[inState, inTape[inHead]]) \\ & \quad \text{then } inHead - 1 \text{ else } inHead + 1 \text{ endif} \\ outState & \equiv \Pi^3_3(prog[inState, inTape[inHead]]) \end{array}$$

Hence we can define the function $\mathsf{execStep}$ by putting

$$\mathsf{execStep} \equiv \lambda prog.\lambda\langle inTape, inHead, inState\rangle.\langle outTape, outHead, outState\rangle$$

In order to define the execution of a Turing machine we only have to apply the function $\mathsf{execStep}$ for an arbitrary number of times. To this aim it is useful to define the function $\mathsf{iterate}$ such that, supposing $a \in A$ and $f \in A \to A$,

$$\begin{cases} \mathsf{iterate}(0, f, a) = a \in A \\ \mathsf{iterate}(\mathsf{s}(x), f, a) = f[\mathsf{iterate}(x, f, a)] \in A \end{cases}$$

i.e. $\mathsf{iterate}(n, f, a) \equiv f^n(a)$. To solve this recursive equation within ITT, we put

$$\mathsf{iterate}(n, f, a) \equiv \mathsf{It}(n)[f, a]$$

so that

$$\begin{cases} \mathsf{It}(0) = \lambda f.\lambda a.a \\ \mathsf{It}(\mathsf{s}(x)) = \lambda f.\lambda a.f[\mathsf{It}(x)[f, a]] \end{cases}$$

which is solved by

$$\mathsf{It}(n) \equiv \mathsf{N}_{rec}(n, \lambda f.\lambda a.a, (u, v)\ \lambda f.\lambda a.f[v[f, a]])$$

Now we can use the function $\mathsf{iterate}$ to obtain

$$(\mathsf{execStep}[prog])^n[\langle initTape, initHead, initState\rangle]$$

i.e. we can calculate the status of the Turing machine after any finite number of steps, but ... see the next chapter.

Chapter 3

The canonical form theorem

3.1 Summary

In this chapter we will prove a *canonical form theorem* for the extensional version of Martin-Löf's type theory with one universe. We choose to prove the theorem for such a version because extensional equality is the most widely used in mathematics and hence this theory is probably the nearest to the usual mathematical practice. Of course, it is possible to extend immediately the proof to a theory which contain both intensional equality (denoted by the inductive type Id in the appendix B) and extensional equality (denoted by the type Eq here and in the appendix B) and even to obtain a full strong normalization result if only intensional equality is present [CCo98]. However we think that the technique that we use here to prove the canonical form theorem and the theory presented are interesting enough to deserve a deep study of a case for which the full normalization theorem does not hold, as we will prove here below[1].

Assume that the empty set N_0 is inhabited, that is, assume that $y : \mathsf{N}_0$. Then, by N_0-elimination over the universe U of the small types, we obtain that

$$\mathsf{R}_0(y) \in \mathsf{Eq}(\mathsf{U}, \mathsf{n}, \mathsf{n} \to \mathsf{n})$$

and hence, by extensional equality elimination, $\mathsf{n} = \mathsf{n} \to \mathsf{n} \in \mathsf{U}$ which yields that

$$\mathsf{N} = \mathsf{N} \to \mathsf{N}$$

that is, we proved that if the empty set is inhabited then the set N of the natural numbers and the set $\mathsf{N} \to \mathsf{N}$ of the functions from the natural numbers into the natural numbers are equal.

Consider now the following derivation[2]

$$\dfrac{\dfrac{\dfrac{\dfrac{y : \mathsf{N}_0}{\vdots}}{\mathsf{N} = \mathsf{N} \to \mathsf{N}} \quad [x : \mathsf{N}]_1}{x \in \mathsf{N} \to \mathsf{N}} \qquad [x : \mathsf{N}]_1}{x[x] \in \mathsf{N}}}{\lambda((x)\ x[x]) \in \mathsf{N} \to \mathsf{N}}\ 1$$

It will be used in the next derivation to prove that a type can be assigned to the term

$$\lambda((x)\ x[x])[\lambda((x)\ x[x])]$$

[1] The proof that is exposed here was suggested to me by B. Nordstrom when we spoke about the possibility for a formal proof of a normalization theorem for Martin-Löf's type theory in late '80. Later, other similar proofs of the same result appeared in the literature; for instance in a personal communication H. Barendreght told me that also S. Berardi obtained a similar result for the calculus of construction (see [Ber90]) and that other examples of not wished consequences of extensional equality can be found in [Geu93].

[2] We write $c[a]$ to denote the application of the function c to the element a.

under the assumption that the empty set is inhabited, while it is well known that this term does have no normal form.

$$\dfrac{\dfrac{\dfrac{}{\lambda((x)\ x[x]) \in \mathsf{N} \to \mathsf{N}} y : \mathsf{N}_0 \quad \dfrac{\dfrac{}{\lambda((x)\ x[x]) \in \mathsf{N} \to \mathsf{N}} \quad \dfrac{}{\mathsf{N} \to \mathsf{N} = \mathsf{N}}}{\lambda((x)\ x[x]) \in \mathsf{N}} y : \mathsf{N}_0}{\lambda((x)\ x[x])[\lambda((x)\ x[x])] \in \mathsf{N}}$$

Note now that we can go a bit further and prove that the closed term

$$\lambda((y)\ \lambda((x)\ x[x])[\lambda((x)\ x[x])])$$

has type

$$\mathsf{N}_0 \to \mathsf{N}$$

and hence we can construct also closed proof which cannot be normalized.

So there is no possibility for a standard kind of normalization theorem for the theory that we are considering. This is the reason why the theorem that we will present in the following does not prove that any typed term can be reduced into an equivalent one in normal form. Nevertheless, it proves that any closed term, namely a term which contains no free variables, such that the derivation of its typebility has no open assumption, can be reduced into an equivalent one in canonical form, that is a sort of external normal form. It can be useful to observe that, since the notion of reduction that we will define will apply only to closed term, and hence non closed terms are not reducible, such a result is not really different from a standard normalization theorem which states that a term can be reduced into an equivalent one which contains no reducible parts. On the other hand, the previous examples of non-normalizing terms show that this is the strongest result that we can hope to obtain.

This result is clearly sufficient to prove that the usual consequences of a standard normalization theorem hold also for this extensional version of Martin-Löf's type theory, namely consistency, disjunction and elimination property and introductory form theorem for any provable judgment.

Of course, due to Gödel second incompleteness theorem, no normalization theorem for a theory as complex as Martin-Löf's type theory is a satisfactory proof of its consistence. Indeed, a *real* consistency argument can be based only on the reliability of the rules of the theory and this cannot stay on any *formal* proof of consistency but only in the meaning of the rules themselves. On the other hand, in our experience there is nothing like the search for a predicative proof of normalization to learn all of the subtleties of a formal system and begin to understand why its rules are safe.

3.2 Introduction

Since the 70's Per Martin-Löf has developed, in a number of successive variants, an Intuitionistic Theory of Types [Mar75, Mar82, Mar84, NPS90]. The initial aim was that of providing a formal system for Constructive Mathematics but the relevance of the theory also in Computer Science was soon recognized. In fact, Martin-Löf's type theory can equally well be viewed as a programming language with a very rich type structure, as a specification language and as an integrated system to derive correct programs from their specifications [NP83, NS84, PS86]. These pleasant properties of the theory have certainly contributed to the interest for it arisen in the computer science community, especially among those people who believe that program correctness is a major concern in the programming activity [BCMS89]. Actually the theory which is quite well known is the one presented in [Mar82, Mar84]. This is the theory we shall consider in this chapter and refer to as Martin-Löf's Intuitionistic Type Theory (ITT), even if successive variations have been developed. Sometime, ITT is referred to as the polymorphic theory opposite to the last version [NPS90] which is monomorphic, i.e. each element can be uniquely typed, and decidable.

In this chapter we shall present an extension of ITT whose principal characteristic consists in the possibility of assuming variables denoting higher order functions. Our main motivation in developing this higher order version (HITT) has been the wish to complete the way first opened

by Per Martin-Löf. Indeed in the preface of [Mar84], while referring to a series of lectures given in Munich (October 1980), he writes: "The main improvement of the Munich lectures, compared with those given in Padova, was the adoption of a systematic higher level notation ...". This notation is called expressions with arity and yields a more uniform and compact writing of the rules of the theory. An expression with arity is built up starting from primitive constants and variables with arity, by means of abstractions and applications. The arity associated to an expression specifies its functionality and constrains the applications, analogously to what the type does for typed lambda-calculus. In our opinion, in order to fully exploit this approach and be able to establish formal properties of the system, it is necessary to extend the formalization of the contexts as given in [Mar84] to assumptions of variables of higher arity. Therefore we have defined this extension that, even if conservative, supplies increased expressive power, advantageous especially when the theory is viewed as a programming and a specification language. In fact, assuming a variable of higher arity corresponds to assuming the possibility of putting together pieces of programs, thus supporting a modular approach in program development [BV87].

Some properties of HITT are also proved, the principal ones are the consistency of HITT, which also implies the consistency of ITT, and the computability of any judgement derived within HITT. Besides we proved a canonical form theorem: to any derivable judgement we can associate a canonical one whose derivation ends with an introductory rule. This result, even if weaker than a standard normalization theorem [Pra65], suffices to obtain all the useful consequences typical of a normal form theorem, mainly the consistency of the theory. Moreover, by using the computational interpretation of types (i.e. types as problem descriptions and their elements as programs to solve those problems) it immediately follows that the execution of any proved correct program terminates. We assume the reader is familiar with the Intuitionistic Theory of Types as presented in [Mar82, Mar84] and with typed lambda calculus [Bar84], or enjoyably, with the theory of expressions with arity (see appendix A or [BV89, NPS90]).

The following is the outline of the chapter. In section 2. our characterization of assumptions of variables of any arity is presented and some consequences of this are briefly sketched. We are extremely grateful to Prof. Aczel for his suggestions on the notation to be used for the new kind of assumptions. Further comments on the properties of our system are given in section 3. Section 4. deals with computability. The definition of computable judgement, which is the basis for the computability theorem, is first given. The rest of the section is devoted to prove that each rule of the theory preserves this property. The computability theorem, as well as some relevant corollaries, is presented in the concluding section 5. All of the rules of HITT are listed in the appendix B. Compared with ITT, besides the changes concerning the assumptions of higher level variables, there are also changes in the notation and in the fact that we explicitly added to the premises of a rule all the requirements that were only informally expressed in ITT.

3.3 Assumptions of high level arity variables

We assume the theory of expressions with arity (see appendix A and [Bee85, BV89, NPS90]) developed by Martin Löf in order to give an uniform and compact presentation of his theory of types. The theory has many similarities with typed lambda-calculus [Bar84] and some familiarity with this system should be sufficient to understand what follows. An expression with arity is built up starting from primitive constants and variables with arity, by means of abstractions and applications. The arity associated to an expression fully specifies its functionality, i.e. it indicates the number and the arity of the expressions to which it can be applied, analogously to what the type does for typed lambda-calculus.

The Intuitionistic Theory of Types [Mar82] consists of a language of constant symbols, each of some arity, a system of computation rules and a system of rules of inference for deriving judgements. Each instance of a rule of inference has the form

$$\frac{J_1 \ldots J_n}{J}$$

where $J_1,\ldots,J_n,J$ are judgements. A derivation is a tree of judgements built up in the usual way

using instances of the rules of inference. Judgements have the form

$$F\ [\Gamma]$$

where Γ is a "context", and F has one of the forms

$$A\ \mathsf{type}$$
$$A = B$$
$$a \in A$$
$$a = b \in A$$

Here A, B, a, b are expressions of arity 0. A context is a list $A_1, \ldots, A_n$ of assumptions where, for $j = 1, \ldots, n$, A_j is an assumption over the context $A_1, \ldots, A_{j-1}$. We will call "order between assumptions condition" this requirement on the assumptions of a context[3]. When the context is empty we write only F instead of $F\ []$, and call J a "closed" judgement as opposed to "hypothetical" judgement, that is with non-empty context. In the following we will say that the context Γ' extends the context Γ if Γ' is obtained from Γ by adding some assumptions satisfying the "order between assumptions" condition. Each assumption has the form

$$x : A\ [\Gamma]$$

where x is a variable of some arity, A is an expression of arity 0 and Γ is a context. We call x the variable of the assumption and its arity is the arity of the assumption. The variables of the assumptions of a context are also called the variables of the context. They must be pair-wise distinct. We will say that the assumption of a variable x depends on all the assumptions of the context. The conditions for forming an assumption over a context involve the notion of derivation, so that the contexts and derivations have to be defined by simultaneous inductive definition. The simple case of an assumption

$$y : B\ [\Gamma]$$

of arity 0 over a context Γ is the familiar one defined by Martin-Löf in the original theory [Mar84]. The conditions are that the judgement $B\ \mathsf{type}\ [\Gamma]$ should be derivable and that y should not be a variable of the context Γ. It is easy to convince ourselves that these conditions are just a formalization of those usually required for making an assumption in a natural deduction system. The variable y keeps the place of a generic object of type B.

To deal with assumptions with arities of higher level we add to the language, for each arity $\alpha \equiv (\alpha_1, \ldots, \alpha_n)$, a constant

$$\mathsf{T}_\alpha$$

of arity

$$(0, (\alpha_1), (\alpha_1)(\alpha_2), \ldots, (\alpha_1)\ldots(\alpha_{n-1}), (\alpha_1)\ldots(\alpha_n))$$

and if $A_1, \ldots, A_n$, $A(x_1, \ldots, x_n)$ are expressions of arity 0 and $x_1, \ldots, x_n$ are distinct variables of arities $\alpha_1, \ldots, \alpha_n$ respectively then we write

$$(x_1 : A_1, \ldots, x_n : A_n)A(x_1, \ldots, x_n)$$

for the expression of arity 0

$$\mathsf{T}_\alpha(A_1, (x_1)A_2, \ldots, (x_1, \ldots, x_{n-1})A_n, (x_1, \ldots, x_n)A(x_1, \ldots, x_n))$$

If

$$(*) \qquad B \equiv (x_1 : A_1, \ldots, x_n : A_n)A(x_1, \ldots, x_n)$$

we write

$$: B\ [\Gamma]$$

for the judgement

$$(**) \qquad A(x_1, \ldots, x_n)\ \mathsf{type}\ [\Gamma, x_1 : A_1, \ldots, x_n : A_n].$$

[3] As in [Bru91] contexts are inserted like 'telescopes'.

When Γ is empty we write only $: B$ instead of $: B\ []$.

We can now state the conditions for forming the assumption

$$y : B\ [\Gamma]$$

of arity

$$\alpha \equiv (\alpha_1, \ldots, \alpha_n)$$

These conditions are that

- $(*)$ holds for some choice of variables $x_1, \ldots, x_n$ not free in B and in Γ and some expressions $A_1, \ldots, A_n$, $A(x_1, \ldots, x_n)$ of arity 0.
- $(**)$ is derivable.
- y is a variable of arity α that is not a variable of the context $\Gamma, x_1 : A_1, \ldots, x_n : A_n$.

As an example suppose

$$x : A\ [\Gamma]$$
$$y : \mathsf{T}(0)(V(x), (v)B(x, v))\ [\Gamma, x : A]$$

are correct assumptions. The variable x keeps the place of a generic object of type A, while the variable y keeps the place of a function from a generic object v of type $V(x)$ to objects of type $B(x, v)$. Now the assumption of a variable z, which keeps the place of a function mapping objects x and functions y to objects of type $C(x, y)$

$$z : \mathsf{T}(0, (0))(A, (x)T(0)(V(x), (v)B(x, v)), (x, y)C(x, y))\ [\Gamma]$$

or, using the abbreviation,

$$z : (x : A, y : (v : V(x))B(x, v))C(x, y)[\Gamma]$$

is correct if the judgement

$$C(x, y)\ \mathsf{type}\ [\Gamma, x : A, y : (v : V(x))B(x, v)]$$

is derivable and z does not occur in the context $[\Gamma, x : A, y : (v : V(x))B(x, v)]$.

In writing the inference rules we will adopt Martin Löf's convention to present explicitly only those assumptions that do not occur in both premises and conclusion. Hence all the assumptions appearing in the premises are to be considered discharged by the application of the rule. Clearly, as usual, the remaining assumptions of the context should not depend on the discharged ones, i.e. they must be an initial segment in the ordered list of assumptions of the context. Moreover we mean that the context of the conclusion of a rule is obtained by merging, without duplication, the contexts (not explicitly present) of the assumptions and (possibly) the assumptions explicitly present in the context of the conclusion.

The assumption rules introduce a new assumption in the context of the conclusion. In order to formulate these rules it is convenient to introduce some abbreviations. When there is a derivation of $: B\ [\Gamma]$ then we use

$$b : B\ [\Gamma]$$

to abbreviate the judgement

$$b(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n)\ [\Gamma, x_1 : A_1, \ldots, x_n : A_n]$$

where the variables $x_1, \ldots, x_n$, of arities $\alpha_1, \ldots, \alpha_n$ respectively, are chosen not free in B and C, so that $(*)$ holds, and b is an expression of arity $(\alpha_1, \ldots, \alpha_n)$ in which they may appear only variables of the context Γ.

Similarly we use

$$b = b' : B\ [\Gamma]$$

to abbreviate the judgement

$$b(x_1,\ldots,x_n) = b'(x_1,\ldots,x_n) \in A(x_1,\ldots,x_n)\ [\Gamma, x_1 : A_1, \ldots, x_n : A_n]$$

Now, if $y : B\ [\Gamma]$ is an assumption, then we have the following assumption rules:

$$\frac{a_1 : \underline{A}_1 \ldots a_n : \underline{A}_n \qquad : B}{y(a_1,\ldots,a_n) \in A(a_1,\ldots,a_n)\ [y : B]}$$

$$\frac{a_1 = a_1' : \underline{A}_1 \ldots a_n = a_n' : \underline{A}_n \qquad : B}{y(a_1,\ldots,a_n) = y(a_1',\ldots,a_n') \in A(a_1,\ldots,a_n)\ [y : B]}$$

where, for $j = 1,\ldots,n$, $\underline{A}_j \equiv ((x_1,\ldots,x_{j-1})A_j)(a_1,\ldots,a_{j-1})$.

Note that, in both cases, in the conclusion appears the new assumption $y : B$ while in the premises there may appear assumptions which are discharged by the rule.

As an example consider again the assumption

$$z : (x : A, y : (v : V(x))B(x,v))C(x,y)[\Gamma]$$

and suppose that the judgements:

1. $: (x : A, y : (v : V(x))B(x,v))C(x,y)\ [\Gamma]$, that is

$$C(x,y)\ \mathsf{type}\ [\Gamma, x : A, y : (v : V(x))B(x,v)]$$

2. $a : A\ [\Gamma_1]$, that is

$$a \in A\ [\Gamma_1]$$

3. $b : (s : V(a))B(a,s)\ [\Gamma_2]$ that is

$$b(s) \in B(a,s)\ [\Gamma_2, s : V(a)]$$

are all derivable judgements, then

$$\frac{a \in A \qquad b(s) \in B(a,s)\ [s : V(a)] \qquad C(x,y)\ \mathsf{type}\ [x : A, y : (v : V(x))B(x,v)]}{z(a,b) \in C(a,b)\ [z : (x : A, y : (v : V(x))B(x,v))C(x,y)]}$$

is an instance of the assumption rule. The context in the conclusion of the rule is the merge, without duplication, of the four contexts Γ, Γ_1, Γ_2 and $z : (x : A, y : (v : V(x))B(x,v))C(x,y)$. The assumptions of the variables s, x and y are discharged by the rule while the assumption of z is possibly introduced.

The given abbreviations for the hypothetical judgements have the nice consequence of allowing a notation quite close to that used by Martin-Löf [Mar84] for variable's substitution, also in the case of high-arity variables. To express the fact that a sequence of variables can be substituted by a given sequence of expressions, we introduce the following concept of fitting substitutions.

Definition 3.3.1 (Fitting substitution) *The sequences of judgements*

$$b_1 : \underline{B}_1\ [\Gamma], \ldots, b_n : \underline{B}_n\ [\Gamma]$$

and

$$b_1 = b_1' : \underline{B}_1\ [\Gamma], \ldots, b_n = b_n' : \underline{B}_n\ [\Gamma]$$

where

$$\underline{B}_i \equiv ((y_1,\ldots,y_{i-1})B_i)(b_1,\ldots,b_{i-1})$$

are substitutions that fit with any context $[\Gamma, y_1 : B_1, \ldots, y_n : B_n]$.

Note that a similar concept of "fitting" is already used in [Bru91] where only variables of arity 0 are considered.

3.4 Modifications due to the new assumptions

Clearly, the new form of assumptions compel us to extend the substitution rules. They are listed in the appendix B among all the other rules but let us analyze an example. The following substitution rule of the previous version [Mar84]

$$\frac{b \in B \qquad d(y) \in D(y)\ [y : B]}{d(b) \in D(b)}$$

which involves a variable y of arity 0, is extended to the new rule

$$\frac{b_1 : \underline{B}_1 \ldots b_n : \underline{B}_n \qquad d(y_1, \ldots, y_n) \in D(y_1, \ldots, y_n)\ [y_1 : B_1, \ldots, y_n : B_n]}{d(b_1, \ldots, b_n) \in D(b_1, \ldots, b_n)}$$

where $\underline{B}_i \equiv ((y_1, \ldots, y_{i-1})B_i)(b_1, \ldots, b_{i-1})$, i.e., $b_1 : \underline{B}_1 \ldots b_n : \underline{B}_n$ is a substitution that fits with the context $[y_1 : B_1, \ldots, y_n : B_n]$[4]. The pattern is rather similar but now the variables $y_1, \ldots, y_n$ may have any arity.

The changes that can be made on other rules are more fundamental. For example, let us analyze the W-elimination rule. In the previous version the W-elimination rule was (adopting our notation)

$$\frac{\begin{array}{c}[x : A, y : \Pi(B(x), (t)\mathsf{W}(A, B)), z : \Pi(B(x), (u)\ C(\mathsf{Ap}(y, u)))]_1 \\ \vdots \\ c \in \mathsf{W}(A, B) \qquad\qquad d(x, y, z) \in C(\mathsf{sup}(x, y))\end{array}}{\mathsf{T}_{rec}(c, d) \in C(c)}\ 1$$

while the new one is

$$\frac{\begin{array}{c}[x : A, y : (t : B(x))\ \mathsf{W}(A, B), z : (u : B(x))\ C(y(u))]_1 \\ \vdots \\ c \in \mathsf{W}(A, B) \qquad\qquad d(x, y, z) \in C(\mathsf{sup}(x, y))\end{array}}{\mathsf{T}_{rec}(c, d) \in C(c)}\ 1$$

which is conceptually more straight. In fact while in the previous version y and z stands for functions (i.e. elements of a Π-type), here they are functional expressions: this avoid the application-abstraction circle, as in $(u)\ C(\mathsf{Ap}(y, u))$, which can now be simply expressed by the application of expressions $C(y(u))$ since it is possible to assume the variable y of arity $(0)0$.

3.5 Some observations on type theory

In the next section we will frequently often use some concepts and properties of HITT that we will briefly describe here.

3.5.1 Associate judgements

Our rules differ from the ones introduced in [Mar84] both for the use of assumptions of variables of higher arity and because when a type A appears in the conclusion of a rule the premises of the rule are augmented with those of the formation rule for the judgement A type. This requirement allows us to easily prove the following theorem 3.5.2 which shows a strong property on the structure of the derivable judgements of HITT. We introduce first the notion of associate judgements.

Definition 3.5.1 (Associate judgements) *The* associate judgement(s) *of*

1)	$a \in A\ [x_1 : A_1, \ldots, x_n : A_n]$	*is*	A type $[x_1 : A_1, \ldots, x_n : A_n]$
2)	$A = B\ [x_1 : A_1, \ldots, x_n : A_n]$	*are*	A type $[x_1 : A_1, \ldots, x_n : A_n]$
		and	B type $[x_1 : A_1, \ldots, x_n : A_n]$
3)	$a = b \in A\ [x_1 : A_1, \ldots, x_1 : A_n]$	*are*	$a \in A\ [x_1 : A_1, \ldots, x_n : A_n]$
		and	$b \in A\ [x_1 : A_1, \ldots, x_n : A_n]$

[4] If no confusion can arise we will use the abbreviation $b :$ or $b = c :$ to denote a substitution that fits with a given context.

Theorem 3.5.2 (Associate judgements derivability) *Let J be a derivable judgement. Then the associate judgements of J are derivable.*

Proof. The three cases should be proved simultaneously. The proof follows almost immediately by induction on the length of the derivation of the considered judgement. Only in some cases structural rules or substitution rules should be carefully used.

Actually, to obtain the previous result it would not be necessary to add in each rule the premises of the formation rule of the judgement A type; for instance, they are superfluous in the Π-introduction rule. We inserted this redundancies for sake of uniformity in view of proving general properties of the theory at a level as much abstract as possible.

3.5.2 Substituted judgements

Substitution is a central operation on judgements. Many concepts that we shall introduce in the next section will be based on the two kinds of substitutions we define now.

Definition 3.5.3 (Tail substituted judgements) *Let $\Delta \equiv [\Gamma, x_1 : A_1, \ldots, x_n : A_n]$ be a context, $a_1 : \underline{A}_1\ [\Gamma]$, $\ldots$, $a_n : \underline{A}_n\ [\Gamma]$ and $a_1 = a'_1 : \underline{A}_1\ [\Gamma]$, $\ldots$, $a_n = a'_n : \underline{A}_n\ [\Gamma]$ be substitutions that fit with the last n assumption in the context Δ, and $J \equiv F\ [\Delta]$ be any judgement. Then*

1. *$J[x_1 := a_1, \ldots, x_n := a_n]$ is an abbreviation for the* tail substituted judgement *of J which is the following:*

   (1.1) $((x_1, \ldots, x_n)\ A)(a_1, \ldots, a_n)$ type $[\Gamma]$
   if $F \equiv A$ type

   (1.2) $((x_1, \ldots, x_n)\ A)(a_1, \ldots, a_n) = ((x_1, \ldots, x_n)\ B)(a_1, \ldots, a_n)\ [\Gamma]$
   if $F \equiv A = B$

   (1.3) $((x_1, \ldots, x_n)\ a)(a_1, \ldots, a_n) \in ((x_1, \ldots, x_n)\ A)(a_1, \ldots, a_n)\ [\Gamma]$
   if $F \equiv a \in A$

   (1.4) $((x_1, \ldots, x_n)\ a)(a_1, \ldots, a_n) = ((x_1, \ldots, x_n)\ b)(a_1, \ldots, a_n) \in ((x_1, \ldots, x_n)\ A)(a_1, \ldots, a_n)\ [\Gamma]$
   if $F \equiv a = b \in A$

2. *$J[x_1 \leftarrow a_1 = a'_1, \ldots, x_n \leftarrow a_n = a'_n]$ is an abbreviation for the* tail substituted judgement *of J which is the following:*

   (2.1) $((x_1, \ldots, x_n)\ A)(a_1, \ldots, a_n) = ((x_1, \ldots, x_n)\ A)(a'_1, \ldots, a'_n)\ [\Gamma]$
   if $F \equiv A$ type

   (2.2) $((x_1, \ldots, x_n)\ A)(a_1, \ldots, a_n) = ((x_1, \ldots, x_n)\ B)(a'_1, \ldots, a'_n)\ [\Gamma]$
   if $F \equiv A = B$

   (2.3) $((x_1, \ldots, x_n)\ a)(a_1, \ldots, a_n) = ((x_1, \ldots, x_n)\ a)(a'_1, \ldots, a'_n) \in ((x_1, \ldots, x_n)\ A)(a_1, \ldots, a_n)\ [\Gamma]$
   if $F \equiv a \in A$

   (2.4) $((x_1, \ldots, x_n)\ a)(a_1, \ldots, a_n) = ((x_1, \ldots, x_n)\ b)(a'_1, \ldots, a'_n) \in ((x_1, \ldots, x_n)\ A)(a_1, \ldots, a_n)\ [\Gamma]$
   if $F \equiv a = b \in A$

The substitutions rules of HITT are sufficient to prove the following theorem.

Theorem 3.5.4 *The tail substituted judgements of a derivable judgement are derivable.*

Proof. Just apply the suitable substitution rule except for the cases (2.2) and (2.4). For these cases first note that if $a_1 = a'_1 : A_1\ [\Gamma]$, $\ldots$, $a_n = a'_n : A_n\ [\Gamma]$ is a substitution that fits with the tail of Δ, then also $a_1 : A_1\ [\Gamma]$, $\ldots$, $a_n : A_n\ [\Gamma]$, whose derivability is showed by theorem 3.5.2, is a substitution that fits with the tail of Δ. Then apply the suitable substitution rule $(:=)$ to $A = B\ [\Delta]$ (case 2.2) or to $a = b \in A\ [\Delta]$ (case 2.4) and the suitable substitution rule $(\leftarrow)$ to B type $[\Delta]$ (case 2.2) or to $b \in A\ [\Delta]$ (case 2.4), which are the associates of $A = B\ [\Delta]$ and $a = b \in A\ [\Delta]$ respectively. The result follows by transitivity.

The second kind of substitution does not rely directly on the substitutions rules. It substitutes the first part of a context and consequentially it modifies not only the form part of the judgement but also the tail of the context.

Definition 3.5.5 (Head substituted judgements) *Let $\Delta \equiv [x_1 : A_1, \ldots, x_n : A_n]$ be a context, $a_1 : A_1$, …, $a_i : A_i$ and $a_1 = a'_1 : A_1$, …, $a_i = a'_i : A_i$ for some $i \leq n$, be substitutions that fit with the first i assumptions of Δ, $J \equiv F\ [\Delta]$ be any judgement. Moreover let $A'_j \equiv ((x_1, \ldots, x_i)\ A_j)(a_1, \ldots, a_i)$ for any $i + 1 \leq j \leq n$. Then*

1. *$J[x_1 := a_1, \ldots, x_i := a_i]$ is an abbreviation for the* head substituted judgement *of J which is the following:*

$$
\begin{array}{ll}
\textit{(1.1)} & ((x_1, \ldots, x_i)\ A)(a_1, \ldots, a_i)\ \mathsf{type}\ [x_{i+1} : A'_{i+1}, \ldots, x_n : A'_n] \\
& \textit{if } F \equiv A\ \mathsf{type} \\
\textit{(1.2)} & ((x_1, \ldots, x_i)\ A)(a_1, \ldots, a_i) = \\
& \qquad ((x_1, \ldots, x_i)\ B)(a_1, \ldots, a_i)[x_{i+1} : A'_{i+1}, \ldots, x_n : A'_n] \\
& \textit{if } F \equiv A = B \\
\textit{(1.3)} & ((x_1, \ldots, x_i)\ a)(a_1, \ldots, a_i) \in \\
& \qquad ((x_1, \ldots, x_i)\ A)(a_1, \ldots, a_i)[x_{i+1} : A'_{i+1}, \ldots, x_n : A'_n] \\
& \textit{if } F \equiv a \in A \\
\textit{(1.4)} & ((x_1, \ldots, x_i)\ a)(a_1, \ldots, a_i) = ((x_1, \ldots, x_i)\ b)(a_1, \ldots, a_i) \in \\
& \qquad ((x_1, \ldots, x_i)\ A)(a_1, \ldots, a_n)[x_{i+1} : A'_{i+1}, \ldots, x_n : A'_n] \\
& \textit{if } F \equiv a = b \in A
\end{array}
$$

2. *$J[x_1 \leftarrow a_1 = a'_1, \ldots, x_i \leftarrow a_i = a'_i]$ is an abbreviation for the* head substituted judgement *of J which is the following:*

$$
\begin{array}{ll}
\textit{(2.1)} & ((x_1, \ldots, x_i)\ A)(a_1, \ldots, a_i) = \\
& \qquad ((x_1, \ldots, x_i)\ A)(a'_1, \ldots, a'_i)[x_{i+1} : A'_{i+1}, \ldots, x_n : A'_n] \\
& \textit{if } F \equiv A\ \mathsf{type} \\
\textit{(2.2)} & ((x_1, \ldots, x_i)\ A)(a_1, \ldots, a_i) = \\
& \qquad ((x_1, \ldots, x_i)\ B)(a'_1, \ldots, a'_i)[x_{i+1} : A'_{i+1}, \ldots, x_n : A'_n] \\
& \textit{if } F \equiv A = B \\
\textit{(2.3)} & ((x_1, \ldots, x_i)\ a)(a_1, \ldots, a_i) = ((x_1, \ldots, x_i)\ a)(a'_1, \ldots, a'_i) \in \\
& \qquad ((x_1, \ldots, x_i)\ A)(a_1, \ldots, a_i)[x_{i+1} : A'_{i+1}, \ldots, x_n : A'_n] \\
& \textit{if } F \equiv a \in A \\
\textit{(2.4)} & ((x_1, \ldots, x_i)\ a)(a_1, \ldots, a_i) = ((x_1, \ldots, x_i)\ b)(a'_1, \ldots, a'_i) \in \\
& \qquad ((x_1, \ldots, x_i)\ A)(a_1, \ldots, a_i)[x_{i+1} : A'_{i+1}, \ldots, x_n : A'_n] \\
& \textit{if } F \equiv a = b \in A
\end{array}
$$

Theorem 3.5.6 *The head substituted judgements of a derivable judgement are derivable.*

Proof. As regard to case (1.) note that if $a_1 : A_1$, …, $a_i : A_i$ is a substitution that fits with $[x_1 : A_1, \ldots, x_i : A_i]$ then $a_1 : A_1$, …, $a_i : A_i$, $x_{i+1} : A_{i+1}$, …, $x_n : A_n$ is a substitution that fits with $[x_1 : A_1, \ldots, x_n : A_n]$. Hence the result follows by using the suitable substitution rule. For case (2.), if $a_1 = a'_1 : A_1$, …, $a_i = a'_i : A_i$ is a substitution that fits with $[x_1 : A_1, \ldots, x_i : A_i]$ then $a_1 = a'_1 : A_1$, …, $a_i = a'_i : A_i$, $x_{i+1} = x_{i+1} : A_{i+1}$, …, $x_n = x_n : A_n$ is a substitution that fits with $[x_1 : A_1, \ldots, x_n : A_n]$. Hence the result follows by using directly the suitable substitution rule except for the sub-cases 2.2 and 2.4 where the associate judgements must be considered (see theorem 3.5.4).

Note that we use the same notation for the head and the tail substitutions since the names of the variables and their positions in the context are sufficient to determine the kind of substitution we want to perform.

3.6 The evaluation tree

In HITT, a set of computation rules is associated to each defined type such as Π, Σ, etc. They specify a process for evaluating expressions denoting elements or types. They apply to variable-free and saturated expressions, that is, expressions of arity 0 in which no variable occurs free. The "normal form" theorem for expressions [BV89], assures us that a variable-free, saturated expression is always definitionally equivalent to an expression of the form $c(a_1, \ldots, a_n)$ where c is a constant. Hence, to evaluate an expression, we first consider its normal form and then detect the suitable computation rule. This can be done by looking at the outermost constant of the expression in normal form and, only in some cases, at the value of its first argument. Then each premise of the selected rule indicate how to continue the process recursively. Clearly, the process of evaluating an expression denoting an element or a type using the computation rules naturally gives rise to a finitary tree: we will refer to it as the *evaluation tree*. Of course an expression evaluates if and only if its evaluation tree is finite. Hence if we know that an expression can be evaluated an induction on the depth of its evaluation tree is a correct proof-method. It can be used to prove the following theorem.

Theorem 3.6.1 *Let c and C be variable-free and saturated expressions. Then*

1. *If $c \Rightarrow g$ then g is a canonical expression for an element, i.e. exactly one of the following holds: $g \equiv \lambda(b)$, $g \equiv \langle a, b\rangle$, $g \equiv \mathsf{inl}(a)$, $g \equiv \mathsf{inr}(b)$, $g \equiv \mathsf{e}$, $g \equiv m_n$, $g \equiv 0$, $g \equiv \mathsf{s}(a)$, $g \equiv \mathsf{sup}(a, b)$, $g \equiv \pi(a, b)$, $g \equiv \sigma(a, b)$, $g \equiv +(a, b)$, $g \equiv \mathsf{eq}(a, b, d)$, $g \equiv \mathsf{n}_n$, $g \equiv \mathsf{n}$, $g \equiv \mathsf{w}(a, b)$.*

2. *If $C \Rrightarrow G$ then G is a canonical expression for a type, i.e. exactly one of the following holds: $G \equiv \Pi(A, B)$, $G \equiv \Sigma(A, B)$, $G \equiv +(A, B)$, $G \equiv \mathsf{Eq}(A, b, d)$, $G \equiv \mathsf{N}_n$, $G \equiv \mathsf{N}$, $G \equiv \mathsf{W}(A, B)$, $G \equiv \mathsf{U}$.*

Note that the objects in the conclusion of a formation rule or an introduction rule are always denoted by canonical expressions. We will call them canonical elements or canonical types respectively. However a canonical expression does not necessarily denote a canonical element or a canonical type. The successive canonical form theorem will certify this whenever we consider judgements derived within the theory. More precisely, if the judgement $a \in A$ (or $A\ \mathsf{type}$) is derived within the theory, then the canonical expression resulting from the evaluation of the expression a (or A) denotes a canonical element (or a canonical type). Moreover, under the same hypothesis, the evaluation process of the expression a (or A) always terminates.

Finally let us also observe that, since the computation rules do not "add" variables, it is obvious that if no variable appears in a (respectively A) and $a \Rightarrow g$ (respectively $A \Rrightarrow G$), then no variable appears in g (respectively G).

3.7 Computability

In this section we introduce the main notions of the chapter: the definitions of computation tree and computable judgement.

To prove a canonical-form theorem for the system we are considering, and whose complete set of rules is reported in the appendix B, we will follow a proof style similar to the one used by Martin-Löf in [Mar71] based on the method of Tait [Tai67] to prove normalization theorems. Therefore we will introduce the notion of computable judgement. This notion applies both to closed judgements and to hypothetical ones. Essentially, to express the computability of a judgement is equivalent to express what it is necessary to know in order to be allowed to formulate that judgement. Hence the definition formally summarizes the meaning of all the forms of judgements which can be obtained by a derivation in type theory. Of course, it is directly inspired by the informal explanation of the rules given in [Mar84], but the needs of formalization make it a very long definition. We will base it on the concept of computation tree which represents the full process needed to recognize the computability of a given judgement. The nodes of a computation tree are labeled by derivable judgements and if J is the label of a node then the labels of its sons are all the judgements whose computability is required in order to establish the computability of J.

As regards hypothetical judgements, their computability is referred to the computability of any closed judgement that can be obtained by substituting, in any possible way, computable judgements to the open assumptions.

As regards closed judgements, the definition obviously differs when considering one form of judgement or another. Still there are some basic common points:

- any term appearing in the judgement must be (syntactically) valuable (evaluation) to a canonical term. This requirement is directly expressed for the two forms A type and $a \in A$ and indirectly, by requiring the computability of the associate judgements (associate), for the forms $A = B$ and $a = b \in A$.
- the equality between a term and its corresponding evaluated form must be a provable judgement (correct evaluation)
- the computability of a judgement is recursively referred to the computability of the components (parts) of the judgement built up with the evaluated canonical terms.

Definition 3.7.1 (Computable judgement) *The judgement $J \equiv F\ [\Gamma]$ is computable if it is derivable and*
Case 1. There is no assumption, i.e. the context Γ is empty.

- *Subcase 1.1: $F \equiv A$ type. Then*
  - *1.1.1 (evaluation) $A \Rrightarrow G_A$*
  - *1.1.2 (correct evaluation) the judgement $A = G_A$ is provable*
  - *1.1.3 (parts) the parts of G_A are computable type(s), i.e.*
    - *if $G_A \equiv \Pi(A_1, A_2)$ then the judgements A_1 type and $A_2(x)$ type $[x : A_1]$ are computable*
    - *if $G_A \equiv \Sigma(A_1, A_2)$ then the judgements A_1 type and $A_2(x)$ type $[x : A_1]$ are computable*
    - *if $G_A \equiv +(A_1, A_2)$ then the judgements A_1 type and A_2 type are computable*
    - *if $G_A \equiv \mathsf{Eq}(A_1, b, d)$ then the judgements A_1 type, $b \in A_1$ and $d \in A_1$ are computable*
    - *if $G_A \equiv \mathsf{N}_n$ then no condition*
    - *if $G_A \equiv \mathsf{N}$ then no condition*
    - *if $G_A \equiv \mathsf{W}(A_1, A_2)$ then the judgements A_1 type and $A_2(x)$ type $[x : A_1]$ are computable*
    - *if $G_A \equiv \mathsf{U}$, i.e. , $A \equiv \mathsf{U}$, then no condition*
- *Subcase 1.2: $F \equiv A = B$ then*
  - *1.2.1 (associate judgements) the associate judgements A type and B type are computable, and hence $A \Rrightarrow G_A$ and $B \Rrightarrow G_B$.*
  - *1.2.2 (parts) G_A and G_B are equal computable types, i.e.*
    - *$G_A \equiv \Pi(A_1, A_2)$ iff $G_B \equiv \Pi(B_1, B_2)$ and the judgements $A_1 = B_1$ and $A_2(x) = B_2(x)$ $[x : A_1]$ are computable*
    - *$G_A \equiv \Sigma(A_1, A_2)$ iff $G_B \equiv \Sigma(B_1, B_2)$ and the judgements $A_1 = B_1$ and $A_2(x) = B_2(x)$ $[x : A_1]$ are computable*
    - *$G_A \equiv +(A_1, A_2)$ iff $G_B \equiv +(B_1, B_2)$ and the judgements $A_1 = B_1$ and $A_2 = B_2$ are computable*
    - *$G_A \equiv \mathsf{Eq}(A_1, a, c)$ iff $G_B \equiv \mathsf{Eq}(B_1, b, d)$ and the judgements $A_1 = B_1$, $a = b \in A_1$ and $c = d \in A_1$ are computable*
    - *$G_A \equiv \mathsf{N}_n$ iff $G_B \equiv \mathsf{N}_n$*
    - *$G_A \equiv \mathsf{N}$ iff $G_B \equiv \mathsf{N}$*
    - *$G_A \equiv \mathsf{W}(A_1, A_2)$ iff $G_B \equiv \mathsf{W}(B_1, B_2)$ and the judgements $A_1 = B_1$ and $A_2(x) = B_2(x)$ $[x : A_1]$ are computable*

    * $G_A \equiv \mathsf{U}$ *iff* $G_B \equiv \mathsf{U}$

- *Subcase 1.3:* $F \equiv c \in A$ *then*
  - *1.3.1 (associate judgements) The associate judgement* A type *is computable, and hence* $A \Rightarrow G_A$
  - *1.3.2 (evaluation)* $c \Rightarrow g$
  - *1.3.3 (correct evaluation)* $c = g \in A$ *is provable*
  - *1.3.4 (parts) the parts of* g *are computable element(s) in* G_A*, i.e.*
    * $G_A \equiv \Pi(A_1, A_2)$ *iff* $g \equiv \lambda(b)$ *and the judgement* $b(x) \in A_2(x)\ [x : A_1]$ *is computable*
    * $G_A \equiv \Sigma(A_1, A_2)$ *iff* $g \equiv \langle a, b\rangle$ *and the judgements* $a \in A_1$ *and* $b \in A_2(a)$ *are computable*
    * $G_A \equiv +(A_1, A_2)$ *iff either* $g \equiv \mathsf{inl}(a)$ *and the judgement* $a \in A_1$ *is computable or* $g \equiv \mathsf{inr}(b)$ *and the judgement* $b \in A_2$ *is computable*
    * $G_A \equiv \mathsf{Eq}(A_1, b, d)$ *iff* $g \equiv r$ *and the judgement* $b = d \in A_1$ *is computable*
    * $G_A \equiv \mathsf{N}_n$ *iff* $g \equiv \mathsf{m}_n$ *for some* $0 \leq m \leq n-1$
    * $G_A \equiv \mathsf{N}$ *iff either* $g \equiv 0$ *or* $g \equiv \mathsf{s}(a)$ *and the judgement* $a \in N$ *is computable*
    * $G_A \equiv \mathsf{W}(A_1, A_2)$ *iff* $g \equiv \mathsf{sup}(a, b)$ *and the judgements* $a \in A_1$ *and* $b(x) \in \mathsf{W}(A_1, A_2)\ [x : A_2(a)]$ *are computable*
    * $G_A \equiv \mathsf{U}$ *iff*
      · *either* $g \equiv \pi(a, b)$ *and the judgements* $a \in \mathsf{U}$ *and* $b(x) \in \mathsf{U}\ [x :< a >]$ *are computable*
      · *or* $g \equiv \sigma(a, b)$ *and the judgements* $a \in \mathsf{U}$ *and* $b(x) \in \mathsf{U}\ [x :< a >]$ *are computable*
      · *or* $g \equiv +(a, b)$ *and the judgements* $a \in \mathsf{U}$ *and* $b \in \mathsf{U}$ *are computable*
      · *or* $g \equiv \mathsf{eq}(a, b, d)$ *and the judgements* $a \in \mathsf{U}$*,* $b \in\!< a >$ *and* $d \in\!< a >$ *are computable*
      · *or* $g \equiv \mathsf{n}_n$
      · *or* $g \equiv \mathsf{n}$
      · *or* $g \equiv \mathsf{w}(a, b)$ *and the judgements* $a \in \mathsf{U}$ *and* $b(x) \in \mathsf{U}\ [x :< a >]$ *are computable*

- *Subcase 1.4:* $F \equiv a = b \in A$ *then*
  - *1.4.1 (associate judgements) the associate judgements* $a \in A$ *and* $b \in A$ *are computable, and hence* $a \Rightarrow g_a$*,* $b \Rightarrow g_b$ *and* $A \Rightarrow G_A$*.*
  - *1.4.2 (parts) the parts of* g_a *and* g_b *are computable equal elements in* G_A*, i.e.*
    * $G_A \equiv \Pi(A_1, A_2)$ *iff* $g_a \equiv \lambda(a')$ *and* $g_b \equiv \lambda(b')$ *and the judgement* $a'(x) = b'(x) \in A_2(x)\ [x : A_1]$ *is computable*
    * $G_A \equiv \Sigma(A_1, A_2)$ *iff* $g_a \equiv \langle a', a''\rangle$ *and* $g_b \equiv \langle b', b''\rangle$ *and the judgements* $a' = b' \in A_1$ *and* $a'' = b'' \in A_2(a')$ *are computable*
    * $G_A \equiv +(A_1, A_2)$ *iff either* $g_a \equiv \mathsf{inl}(a')$ *and* $g_b \equiv \mathsf{inl}(b')$ *and the judgement* $a' = b' \in A_1$ *is computable or* $g_a \equiv \mathsf{inr}(a'')$ *and* $g_b \equiv \mathsf{inr}(b'')$ *and the judgement* $a'' = b'' \in A_2$ *is computable*
    * $G_A \equiv \mathsf{Eq}(A_1, c, d)$ *iff* $g_a \equiv r$ *and* $g_b \equiv r$ *and the judgement* $c = d \in A_1$ *is computable*
    * $G_A \equiv \mathsf{N}_n$ *iff* $g_a \equiv \mathsf{m}_n$ *and* $g_b \equiv \mathsf{m}_n$ *for some* $0 \leq m \leq n-1$
    * $G_A \equiv \mathsf{N}$ *iff either* $g_a \equiv 0$ *and* $g_b \equiv 0$ *or* $g_a \equiv \mathsf{s}(a')$ *and* $g_b \equiv \mathsf{s}(b')$ *and the judgement* $a' = b' \in \mathsf{N}$ *is computable*
    * $G_A \equiv \mathsf{W}(A_1, A_2)$ *iff* $g_a \equiv \mathsf{sup}(a', a'')$ *and* $g_b \equiv \mathsf{sup}(b', b'')$ *and the judgements* $a' = b' \in A_1$ *and* $a''(x) = b''(x) \in \mathsf{W}(A_1, A_2)\ [x : A_2(a')]$ *are computable*

* $G_A \equiv \mathsf{U}$ *iff*
  - *either* $g_a \equiv \pi(a', a'')$ *and* $g_b \equiv \pi(b', b'')$ *and the judgements* $a' = b' \in \mathsf{U}$ *and* $a''(x) = b''(x) \in \mathsf{U}\ [x :< a' >]$ *are computable*
  - *or* $g_a \equiv \sigma(a', a'')$ *and* $g_b \equiv \sigma(b', b'')$ *and the judgements* $a' = b' \in \mathsf{U}$ *and* $a''(x) = b''(x) \in \mathsf{U}\ [x :< a' >]$ *are computable*
  - *or* $g_a \equiv +(a', a'')$ *and* $g_b \equiv +(b', b'')$ *and the judgements* $a' = b' \in \mathsf{U}$ *and* $a'' = b'' \in \mathsf{U}$ *are computable*
  - *or* $g_a \equiv \mathsf{eq}(a', c, d)$ *and* $g_b \equiv \mathsf{eq}(b', e, f)$ *and the judgements* $a' = b' \in \mathsf{U}$, $c = e \in< a' >$ *and* $d = f \in< a' >$ *are computable*
  - *or* $g_a \equiv \mathsf{n}_n$ *and* $g_b \equiv \mathsf{n}_n$
  - *or* $g_a \equiv \mathsf{n}$ *and* $g_b \mathsf{n}$
  - *or* $g_a \equiv \mathsf{w}(a', a'')$ *and* $g_b \equiv \mathsf{w}(b', b'')$ *and the judgements* $a' = b' \in \mathsf{U}$ *and* $a''(x) = b''(x) \in \mathsf{U}\ [x :< a' >]$ *are computable*

Case 2. There are assumptions, i.e. $\Gamma \equiv x_1 : A_1, \ldots, x_n : A_n$, *for some* $n > 0$. *The judgement* J *is computable if for any computable closed substitution (c.c.s.)* $a_1 : A_1$, $\ldots$, $a_n : A_n$ *(i.e.* $a_i : A_i$, *for* $1 \leq i \leq n$, *are computable judgements), and for any computable closed substitution (c.c.s.)* $a_1 = c_1 : A_1$, $\ldots$, $a_n = c_n : A_n$ *(i.e.* $a_i = c_i : A_i$, *for* $1 \leq i \leq n$, *are computable judgements) that fit with* Γ

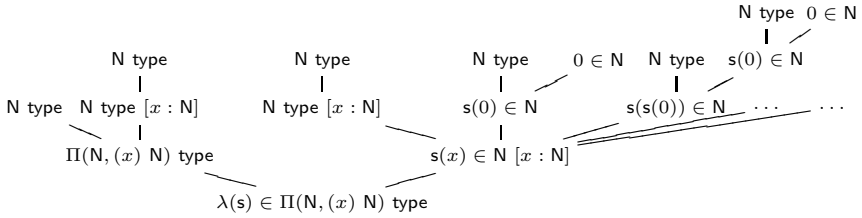
- *Subcase 2.1:* $F \equiv B(x_1, \ldots, x_n)$ type
  - *2.1.1 (substitution* $:=$*) the judgement* $B(a_1, \ldots, a_n)$ type *is computable*
  - *2.1.2 (substitution* $\leftarrow$*) the judgement* $B(a_1, \ldots, a_n) = B(c_1, \ldots, c_n)$ *is computable*
- *Subcase 2.2:* $F \equiv B(x_1, \ldots, x_n) = D(x_1, \ldots, x_n)$ *then*
  - *2.2.1 (associate) the judgement* $B(x_1, \ldots, x_n)$ type $[\Gamma]$ *is computable*
  - *2.2.2 (substitution* $:=$*) the judgement* $B(a_1, \ldots, a_n) = D(a_1, \ldots, a_n)$ *is computable*
  - *2.2.3 (substitution* $\leftarrow$*) the judgement* $B(a_1, \ldots, a_n) = D(c_1, \ldots, c_n)$ *is computable*
- *Subcase 2.3:* $F \equiv b(x_1, \ldots, x_n) \in B(x_1, \ldots, x_n)$ *then*
  - *2.3.1 (associate) the judgement* $B(x_1, \ldots, x_n)$ type $[\Gamma]$ *is computable*
  - *2.3.2 (substitution* $:=$*) the judgement* $b(a_1, \ldots, a_n) \in B(a_1, \ldots, a_n)$ *is computable*
  - *2.3.3 (substitution* $\leftarrow$*) the judgement* $b(a_1, \ldots, a_n) = b(c_1, \ldots, c_n) \in B(a_1, \ldots, a_n)$ *is computable*
- *Subcase 2.4:* $F \equiv b(x_1, \ldots, x_n) = d(x_1, \ldots, x_n) \in B(x_1, \ldots, x_n)$ *then*
  - *2.4.1 (associate) the judgement* $b(x_1, \ldots, x_n) \in B(x_1, \ldots, x_n)$ $[\Gamma]$ *is computable*
  - *2.4.2 (substitution* $:=$*) the judgement* $b(a_1, \ldots, a_n) = d(a_1, \ldots, a_n) \in B(a_1, \ldots, a_n)$ *is computable*
  - *2.4.3 (substitution* $\leftarrow$*) the judgement* $b(a_1, \ldots, a_n) = d(c_1, \ldots, c_n) \in B(a_1, \ldots, a_n)$ *is computable*

Note that the asymmetry in the conditions on associate judgements (point 2.2.1 and 2.4.1) reflects the asymmetry in the rules of the theory. Actually we will prove that also the other associate judgement is computable but the reduced requirement simplifies the next inductive proofs.

By looking at the above definition as a "generalized process" to search for computability of a judgement, a search tree is naturally associate to any derivable judgement. It is clear that whenever J is recognized to be a computable judgement its search tree is well founded. In such a case we give the definition of computation tree.

Definition 3.7.2 (Computation tree) *The computation tree of the computable judgement J is a tree whose root is J and whose principal sub-trees are the computation trees of all the judgements whose computability is required to prove that J is computable.*

For instance, the computable judgement $\lambda(\mathsf{s}) \in \Pi(\mathsf{N}, (x)\mathsf{N})$ has the following computation tree:



In general the computation tree of a judgement J is an infinitary tree: a node has a finite number of branches when we deal with closed judgements, and this number is related to the parts, and a possibly infinite one when we deal with hypothetical judgements.

Note that if we know that a judgement is computable the use of induction on the complexity of its well founded computation tree is a correct proof-method.

Definition 3.7.3 (Computational complexity) *Let J be a computable judgement. We will call* computational complexity *of J the ordinal which measures the complexity of its computation tree T, in the following way:*

$$\begin{cases} 0 & \text{if } T \text{ is a leaf} \\ \bigvee_{i \in I}(\alpha_i + 1) & \text{if } T \text{ has principal sub-trees } T_i \\ & \text{of computational complexity } \alpha_i,\ i \in I \end{cases}$$

We will use both the notation "$\mathsf{comp}(J) = \beta$" and the notation "J comp β" to mean that J is a computable judgement of computational complexity β.

3.8 The lemmas

We are now going to prove that any judgement derivable in the theory is computable. The proof will consist in proving that each rule preserves computability, that is, if the judgements in the premises of a rule are computable then also the judgement in the conclusion of the rule is computable. Of course, this is the inductive step in a proof by induction on the depth of the derivation of the considered judgement.

Note that the computability of the judgements in the base cases is given by definition. Generally, the inductive step for a particular rule will be carried on by subordinate induction on the computational complexity of one of the judgements appearing in the premises of the rule, usually the first one which has no assumption discharged.

We will consider only "full-context" derivations, i.e. derivations build up by applying a rule only if the assumptions which are not discharged by the rule are equal in all the premises, with the only exception of the assumption rules. Note that this is not restrictive since every derivable judgement can be derived by a full-context derivation.

Before starting this analysis of the rules we state some results which follow rather directly from the definition of computable judgement and which are useful in simplifying the subsequent lemmas.

Proposition 3.8.1 (N_0 is empty) *The closed judgement $c \in \mathsf{N}_0$ is not computable.*

It is stated by the definition of computable judgement since there are no canonical elements in N_0.

Proposition 3.8.2 *Every hypothetical judgement with open assumption $x : \mathsf{N}_0$ is computable.*

Proposition 3.8.3 (Evaluation-free) *The following relation between computational complexity of computable judgements hold.*

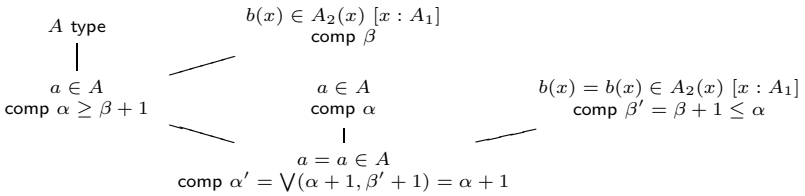


Figure 3.1: reflexivity on elements: case $\Gamma = \emptyset$

1. *If* A type comp β *and* $A \Rrightarrow G_A$ *then* G_A type comp β.
2. *If* $A = C$ comp β *and* $A \Rrightarrow G_A$ *and* $C \Rrightarrow G_C$ *then* $G_A = G_C$ comp β.
3. *If* $a \in A$ comp β *and* $a \Rightarrow g_a$ *and* $A \Rrightarrow G_A$ *then* $g_a \in G_A$ comp β.
4. *If* $a = b \in A$ comp β *and* $a \Rightarrow g_a$, $b \Rightarrow g_b$ *and* $A \Rrightarrow G_A$ *then* $g_a = g_b \in G_A$ comp β.

We will conclude this subsection with the analysis of the simplest rules; we establish also some direct consequences of the definition of computable judgement.

Lemma 3.8.4 (Weakening rules) *If* F $[\Gamma]$ *is computable then, for any context* Γ' *extending* Γ, F $[\Gamma']$ *is computable.*

Proof. When considering associate judgements, if any, the claim follows by induction on the computational complexity of F $[\Gamma]$. When considering substitutions just observe that any c.c.s. that fits with Γ' fits also with Γ, with redundancies, and we yet know that the resulting substituted judgement is computable.

The next lemma on the reflexivity rule states not only that the rule preserves computability but gives us also a relation between the computational complexities of the judgements in the premise and in the conclusion of the rule. This kind of information, on the dependencies among the computational complexities of computable judgements, has a crucial role in the successive proofs when we proceed by induction on the computational complexity of a judgement. The dependencies are often easy to determine simply by looking at the computation tree of one of the considered judgements.

Lemma 3.8.5 (Reflexivity on elements) *The* reflexivity on elements *rule preserves computability, that is, if*

$$a \in A \ [\Gamma] \text{ comp } \alpha$$

then

$$a = a \in A \ [\Gamma] \text{ comp } \alpha + 1$$

Proof. By induction on the computational complexity of the computable judgement $a \in A$ $[\Gamma]$. Subcase $\Gamma = \emptyset$. The associate judgements are computable by hypothesis. To prove the computability of the parts we should analyze each possible form of the values of a and A. Let us consider only the case $a \Rightarrow \lambda(b)$ and $A \Rrightarrow \Pi(A_1, A_2)$. Fig. 3.1 illustrates a piece of the computation tree for this case. There is only one part judgement, that is $b(x) = b(x) \in A_2(x)$ $[x : A_1]$. Its computability follows, by inductive hypothesis, from the computability of the judgement $b(x) \in A_2(x)$ $[x : A_1]$. Thus, $a = a \in A$ is computable. It remains to prove the stated relations on complexities. Let α, α', β, β' be the computational complexities of the computable judgements $a \in A$, $a = a \in A$, $b(x) \in A_2(x)$ $[x : A_1]$, $b(x) = b(x) \in A_2(x)$ $[x : A_1]$. The computability of $a \in A$ depends on the computability of $b(x) \in A_2(x)$ $[x : A_1]$, then we have $\alpha \geq \beta + 1$. By applying the inductive hypothesis we have $\beta' = \beta + 1$, and hence $\alpha' = \bigvee(\alpha + 1, \beta' + 1) = \alpha + 1$. For all the other cases the proof is analogous.

Subcase $\Gamma \neq \emptyset$. The computability of the associate judgement of $a = a \in A$ $[\Gamma]$ is given by hypothesis, while that of its substituted judgements directly follows by inductive hypothesis. Moreover, if

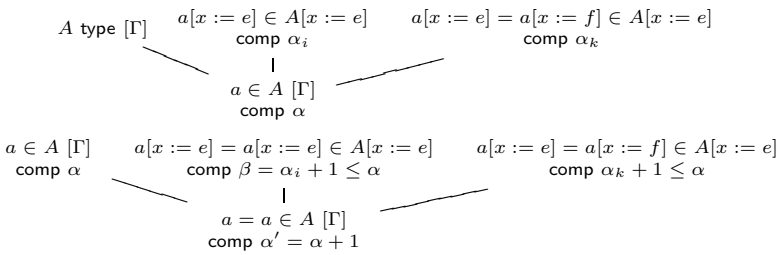


Figure 3.2: reflexivity on elements: case $\Gamma \neq \emptyset$

$\mathsf{comp}(a[x := e] = a[x := e] \in A[x := e]) = \beta$ and $\mathsf{comp}(a[x := e] \in A[x := e]) = \alpha_i$ then, by inductive hypothesis, $\beta = \alpha_i + 1$, hence $\beta + 1 \leq \alpha + 1$ since $a \geq \alpha_i + 1$, and $\alpha' = \bigvee(\alpha + 1, \beta + 1) = \alpha + 1$. See the Fig. 3.2.

Lemma 3.8.6 (Reflexivity on types) *The* reflexivity on types *rule preserves computability, i.e. if*

$$A \text{ type } [\Gamma] \text{ comp } \alpha$$

then

$$A = A \ [\Gamma] \text{ comp } \alpha' = \alpha + 1$$

Proof. The proof, by induction on the computational complexity of the computable judgement A type $[\Gamma]$, is analogous to the one of the previous lemma 3.8.5 except when the value of A is $\mathsf{Eq}(A_1, a, b)$ where the use of the reflexivity-on-elements lemma is needed.

The next corollary is a purely technical result that we will use in the next lemmas to make the proofs shorter.

Corollary 3.8.7 *The following statement on the computability of judgements hold:*

1. *Let $a \in A$ and $c \in A$ be computable closed judgements and g be the value both of a and c. Then, if $a = c \in A$ is derivable then it is also computable.*
2. *Let A type and C type be computable closed judgements and G be the value both of A and C. Then, if $A = C$ is derivable then it is also computable.*
3. *Let $a \in A$ be a computable closed judgement and g be the value of a; then $a = g \in A$ is computable.*
4. *Let A type be a computable closed judgement and G be the value of A; then $A = G$ is computable.*

Proof. Let us prove the various points one after the other.

1. The associate judgements are computable by hypothesis, then we must only prove that the parts are computable. Since $a \in A$ is computable and $a \Rightarrow g$, if $A \Rrightarrow G$ then the judgement $g \in G$ is computable, by proposition 3.8.3 (point 3). Hence, by the reflexivity-on-element lemma, also the judgement $g = g \in G$ is computable and hence the parts of g and g are equal computable elements in G.
2. The proof is analogous to point (1) except for the use of point 1 of proposition 3.8.3, instead of point 3, and the use of the reflexivity-on-type lemma, instead of the reflexivity-on-element lemma.
3. The proof follows by point (1) if we prove that $g \in A$ is computable. By correct evaluation, the judgement $a = g \in A$ is derivable and hence also its associate judgement $g \in A$ is derivable. Its computability then follows by reflexivity and the fact that the parts of $g \in A$ are exactly those of $a \in A$ which is computable by hypothesis.

4. The proof is analogous to point (3) except for the use of point (2) instead of (1).

The following lemma does not concern one of the rules of the theory but states some properties of computable judgements which will be very often referred to in the following subsections.

Lemma 3.8.8 (Head substitution) *Let $\Gamma \equiv [x_1 : A_1, \ldots, x_n : A_n]$ be a context, $J \equiv F\ [\Gamma]$ be a computable judgement, $a_1 : A_1$, ..., $a_i : A_i$ and $a_1 = a'_1 : A_1$, ..., $a_i = a'_i : A_i$, for $i < n$, be c.c.s that fit with the context $[x_1 : A_1, \ldots, x_i : A_i]$. Then*

1. *$J[x_1 := a_1, \ldots, x_i := a_i]$ is a computable judgement*[5].
2. *$J[x_1 \leftarrow a_1 = a'_1, \ldots, x_i \leftarrow a_i = a'_i]$ is a computable judgement.*

Proof. The proof is by induction on the computational complexity of J.

Let $\Delta \equiv [x_{i+1} : A'_{i+1}, \ldots, x_n : A'_n]$, where $A'_j \equiv ((x_1, \ldots, x_i)\ A_j)(a_1, \ldots, a_i)$, for $i+1 \leq j \leq n$, and let $a_{i+1} : A'_{i+1}$, ..., $a_n : A'_n$ be a c.c.s. that fits with the context Δ.

To prove the computability of the head substituted judgements we will show that for any c.c.s. saturating $J[x_1 := a_1, \ldots, x_i := a_i]$ or $J[x_1 \leftarrow a_1 = a'_1, \ldots, x_i \leftarrow a_i = a'_i]$ it is possible to find out a c.c.s. saturating J and yielding the same judgement. First of all note that for $i+1 \leq j \leq n$,

$$\begin{array}{lcl} A_j & \equiv & ((x_1, \ldots, x_{j-1})\ A_j)(a_1, \ldots, a_{j-1}) \\ & \equiv & ((x_{i+1}, \ldots, x_{j-1})(((x_1, \ldots, x_i)\ A_j)(a_1, \ldots, a_i))(a_{i+1}, \ldots, a_{j-1}) \\ & \equiv & A'_j. \end{array}$$

Case i.

(associate judgements) The computability of the associate judgements, if any, follows by inductive hypothesis.

(substitution :=) For any c.c.s. $a_{i+1} : A'_{i+1}$, ..., $a_n : A'_n$ we have that $a_1 : A_1$, ..., $a_n : A_n$ is a c.c.s. that fits with Γ; hence

$$(J[x_1 := a_1, \ldots, x_i := a_i])[x_{i+1} := a_{i+1}, \ldots, x_n := a_n] \equiv J[x_1 := a_1, \ldots, x_n := a_n]$$

is computable.

(substitution $\leftarrow$) For any c.c.s. $a_{i+1} = a'_{i+1} : A'_{i+1}$, ..., $a_n = a'_n : A'_n$ that fits with the context Δ, we have that $a_1 = a_1 : A_1$, ..., $a_i = a_i : A_i$, $a_{i+1} = a'_{i+1} : A_{i+1}$, ..., $a_n = a'_n : A_n$ is a c.c.s. that fits with Γ. Note that the reflexivity-on-elements lemma must be used here. Hence

$$\begin{array}{l} (J[x_1 := a_1, \ldots, x_i := a_i])[x_{i+1} \leftarrow a_{i+1} = a'_{i+1}, \ldots, x_n \leftarrow a_n = a'_n] \\ \equiv J[x_1 \leftarrow a_1 = a_1, \ldots, x_i \leftarrow a_i = a_i, x_{i+1} \leftarrow a_{i+1} = a'_{i+1}, \ldots, x_n \leftarrow a_n = a'_n] \end{array}$$

is computable.

Case ii.

(associate judgements) The computability of the associate judgements follows from case (i) since if $a_1 = a'_1 : A_1$, ..., $a_i = a'_i : A_i$, for $i < n$, are computable then also $a_1 : A_1$, ..., $a_i : A_i$ are computable and

$$J[x_1 := a_1, \ldots, x_i := a_i]$$

is the associate judgement of

$$J[x_1 \leftarrow a_1 = a'_1, \ldots, x_i \leftarrow a_i = a'_i]$$

whose computability is required.

(substitution :=) For any c.c.s. $a_{i+1} : A'_{i+1}$, ..., $a_n : A'_n$ that fits with the context Δ, we have that $a_1 = a'_1 : A_1$, ..., $a_i = a'_i : A_i$, $a_{i+1} = a_{i+1} : A_{i+1}$, ..., $a_n = a_n : A_n$ is a c.c.s. that fits with Γ; hence also

$$\begin{array}{l} (J[x_1 \leftarrow a_1 = a'_1, \ldots, x_i \leftarrow a_i = a'_i])[x_{i+1} := a_{i+1}, \ldots, x_n := a_n] \\ \equiv J[x_1 \leftarrow a_1 = a'_1, \ldots, x_i \leftarrow a_i = a'_i, x_{i+1} \leftarrow a_{i+1} = a_{i+1}, , x_n \leftarrow a_n = a_n] \end{array}$$

[5] Note that for i=n the claim is true by definition of computable judgement.

is computable.

(substitution $\leftarrow$) For any c.c.s. $a_{i+1} = a'_{i+1} : A'_{i+1}, \ldots, a_n = a'_n : A'_n$ that fits with the context Δ, we have that $a_1 = a'_1 : A_1, \ldots, a_i = a'_i : A_i, a_{i+1} = a'_{i+1} : A_{i+1}, \ldots, a_n = a'_n : A_n$ is a c.c.s. that fits with Γ; hence

$$(J[x_1 \leftarrow a_1 = a'_1, \ldots, x_i \leftarrow a_i = a'_i])[x_{i+1} \leftarrow a_{i+1} = a'_{i+1}, \ldots, x_n \leftarrow a_n = a'_n]$$
$$\equiv J[x_1 \leftarrow a_1 = a'_1, \ldots, x_i \leftarrow a_i = a'_i, x_{i+1} \leftarrow a_{i+1} = a'_{i+1}, \ldots, x_n \leftarrow a_n = a'_n]$$

is computable.

Remark 3.8.9 *Let $\Gamma \equiv [x_1 : A_1, \ldots, x_n : A_n]$ be a context, $a_1 : A_1, \ldots, a_n : A_n$ and $a_1 = a'_1 : A_1, \ldots, a_n = a'_n : A_n$ be c.c.s.s that fit with Γ, and $B \equiv (s_1 : S_1, \ldots, s_m : S_m)\ A(s_1, \ldots, s_m)$, then from the head substitution lemma we have that*

- *if $: B\ [\Gamma]$ is computable then*
  $: B\ [\Gamma][x_1 := a_1, \ldots, x_n := a_n]$ and
  $: B\ [\Gamma][x_1 \leftarrow a_1 = a'_1, \ldots, x_n \leftarrow a_n = a'_n]$ are computable;
- *if $b : B\ [\Gamma]$ is computable then*
  $b : B\ [\Gamma][x_1 := a_1, \ldots, x_n := a_n]$ and
  $b : B\ [\Gamma][x_1 \leftarrow a_1 = a'_1, \ldots, x_n \leftarrow a_n = a'_n]$ are computable;
- *if $b = b' : B\ [\Gamma]$ is computable then*
  $b = b' : B\ [\Gamma][x_1 := a_1, \ldots, x_n := a_n]$ and
  $b = b' : B\ [\Gamma][x_1 \leftarrow a_1 = a'_1, \ldots, x_n \leftarrow a_n = a'_n]$ are computable.

We will continue now by proving that each rule listed in the appendix B preserves computability, that is any judgement in the conclusion of that rule is computable whenever all the judgements in the premises are computable. The ordering of the lemmas has been suitably chosen to allow us to deal separately with each rule, thus mastering the complexity of the computability proof.

3.9 Computability of the rules

In the following sections we will show that all of the rules of type theory preserve computability.

3.9.1 The substitution rules

The definition of computable judgement directly states that the substitution rules preserve computability in the special case of saturating substitutions. In the next lemma we will prove that computability is preserved by substitution rules also in the general case of tail substitution. Since the different forms of judgement of the six substitution rules are not essential to prove the result we will compact the sentence as much as possible.

Lemma 3.9.1 (Substitution lemma) *Let Γ be a context, $\Delta \equiv [\Gamma, x_1 : A_1, \ldots, x_n : A_n]$ be a context, $J \equiv F\ [\Delta]$ be a computable judgement, $a_1 : B_1\ [\Gamma], \ldots, a_n : B_n\ [\Gamma]$ and $a_1 = a'_1 : B_1\ [\Gamma], \ldots, a_n = a'_n : B_n\ [\Gamma]$ be two lists of substitutions that fit with the context $[x_1 : A_1, \ldots, x_n : A_n]$, that is, for any $1 \leq j \leq n$, $B_j \equiv ((x_1, \ldots, x_{j-1})\ A_j)(a_1, \ldots, a_{j-1})$. Then*

1. *$J[x_1 := a_1, \ldots, x_n := a_n]$ is a computable judgement.*
2. *$J[x_1 \leftarrow a_1 = a'_1, \ldots, x_n \leftarrow a_n = a'_n]$ is a computable judgement.*

Proof. If the context Γ is empty then the claim holds by definition. Thus, let us suppose that $\Gamma \equiv [s_1 : S_1, \ldots, s_m : S_m]$, for some $m > 0$. The proof is by induction on the computational complexity of J.

Case i.

(associate judgements) The computability of the associate judgements follows by inductive hypothesis.

(substitution :=) For any c.c.s. $c_1 : S_1, \ldots, c_m : S_m$ fitting with the context Γ, we define

$$d_i \equiv ((s_1, \ldots, s_m)\ a_i)(c_1, \ldots, c_m) \qquad 1 \leq i \leq n$$

By remark 3.8.9

$$d_i : ((s_1, \ldots, s_m)\ B_i)(c_1, \ldots, c_m) \equiv a_i : B_i\ [\Gamma][s_1 := c_1, \ldots, s_m := c_m]$$

is computable. Moreover we have

$$\begin{array}{ll} \equiv & ((s_1, \ldots, s_m)\ B_i)(c_1, \ldots, c_m) \\ \equiv & ((s_1, \ldots, s_m)((x_1, \ldots, x_{i-1})\ A_i)(a_1, \ldots, a_{i-1}))(c_1, \ldots, c_m) \\ \equiv & ((s_1, \ldots, s_m, x_1, \ldots, x_{i-1})\ A_i)(c_1, \ldots, c_m, d_1, \ldots, d_{i-1}) \\ \equiv & A_i \end{array}$$

Therefore $c_1 : S_1$, ..., $c_m : S_m$, $d_1 : A_1$, ..., $d_n : A_n$ is a c.c.s. fitting with the context Δ. Hence

$$\begin{array}{l} (J[x_1 := a_1, \ldots, x_n := a_n])[s_1 := c_1, \ldots, s_m := c_m] \equiv \\ \qquad J[s_1 := c_1, \ldots, s_m := c_m, x_1 := d_1, \ldots, x_n := d_n] \end{array}$$

is computable.

(substitution $\leftarrow$) For any c.c.s. $c_1 = c_1' : S_1$, ..., $c_m = c_m' : S_m$ that fits with the context Γ, we define

$$d_i \equiv ((s_1, \ldots, s_m)\ a_i)(c_1, \ldots, c_m) \qquad 1 \leq i \leq n$$

and

$$d_i' \equiv ((s_1, \ldots, s_m)\ a_i)(c_1', \ldots, c_m') \qquad 1 \leq i \leq n$$

By remark 3.8.9

$$d_i = d_i' : ((s_1, \ldots, s_m)\ B_i)(c_1, \ldots, c_m) \equiv a_i : B_i\ [\Gamma][s_1 \leftarrow c_1 = c_1', \ldots, s_m \leftarrow c_m = c_m'] \quad 1 \leq i \leq n$$

is computable. Moreover from the previous point we have

$$A_i \equiv ((s_1, \ldots, s_m)\ B_i)(c_1, \ldots, c_m)$$

Then, $c_1 = c_1' : S_1$, ..., $c_m = c_m' : S_m$, $d_1 = d_1' : A_1$, ..., $d_n = d_n' : A_n$ is a c.c.s. that fits with the context Δ. Hence

$$\begin{array}{l} (J[x_1 := a_1, \ldots, x_n := a_n])[s_1 \leftarrow c_1 = c_1', \ldots, s_m \leftarrow c_m = c_m'] \equiv \\ \qquad J[s_1 \leftarrow c_1 = c_1', \ldots, s_m \leftarrow c_m = c_m', x_1 \leftarrow d_1 = d_1', \ldots, x_n \leftarrow d_n = d_n'] \end{array}$$

is computable.

Case ii.

(associate judgements) The computability of the associate judgements follows from case (i).

(substitution :=) For any c.c.s. $c_1 : S_1, \ldots, c_m : S_m$ fitting with the context Γ, we define

$$d_i \equiv ((s_1, \ldots, s_m)\ a_i)(c_1, \ldots, c_m)$$

and

$$d_i' \equiv ((s_1, \ldots, s_m)\ a_i')(c_1, \ldots, c_m)$$

By remark 3.8.9

$$d_i = d_i' : ((s_1, \ldots, s_m)\ B_i)(c_1, \ldots, c_m) \equiv a_i = a_i' : B_i\ [\Gamma][s_1 := c_1, \ldots, s_m := c_m]$$

is computable. Then, since $A_i \equiv ((s_1, \ldots, s_m)\ B_i)(c_1, \ldots, c_m)$, $c_1 = c_1 : S_1$, ..., $c_m = c_m : S_m$, $d_1 = d_1' : A_1$, ..., $d_n = d_n' : A_n$ is a c.c.s. fitting with the context Δ. Hence

$$\begin{array}{l} (J[x_1 \leftarrow a_1 = a_1', \ldots, x_n \leftarrow a_n = a_n'])[s_1 := c_1, \ldots, s_m := c_m] \equiv \\ \qquad J[s_1 \leftarrow c_1 = c_1, \ldots, s_m \leftarrow c_m = c_m, x_1 \leftarrow d_1 = d_1', \ldots, x_n \leftarrow d_n = d_n'] \end{array}$$

is computable.

(substitution $\leftarrow$) For any c.c.s. $c_1 = c_1' : S_1, \ldots, c_m = c_m' : S_m$ fitting with the context Γ, we define

$$d_i \equiv ((s_1, \ldots, s_m)\ a_i)(c_1, \ldots, c_m)$$

and

$$d_i' \equiv ((s_1, \ldots, s_m)\ a_i)(c_1', \ldots, c_m')$$

By remark 3.8.9

$$d_1 = d_i' : ((s_1, \ldots, s_m)\ B_i)(c_1, \ldots, c_m) \equiv a_i = a_i' : B_i\ [\Gamma][s_1 \leftarrow c_1 = c_1', \ldots, s_m \leftarrow c_m = c_m']$$

is computable. Then, since $A_i \equiv ((s_1, \ldots, s_m)\ B_i)(c_1, \ldots, c_m)$, $c_1 = c_1' : S_1, \ldots, c_m = c_m' : S_m$, $d_1 = d_1' : A_1, \ldots, d_n = d_n' : A_n$ is a c.c.s. fitting with the context Δ. Hence

$$\begin{aligned}(J[x_1 \leftarrow a_1 = a_1', \ldots, x_n \leftarrow a_n = a_n'])[s_1 \leftarrow c_1 = c_1', \ldots, s_m \leftarrow c_m = c_m'] \equiv \\ J[s_1 \leftarrow c_1 = c_1', \ldots, s_m \leftarrow c_m = c_m', x_1 \leftarrow d_1 = d_1', .., x_n \leftarrow d_n = d_n']\end{aligned}$$

is computable.

3.9.2 U-elimination rules

The next lemma 3.9.2 deals with U-elimination rules. We need to know that they preserve both computability and computational complexity to establish the next lemma 3.9.3 about computability of the remaining structural rules. For this reason their analysis precede so much that of all the other logical rules.

Lemma 3.9.2 (U-elimination) *The* U*-elimination rules preserve computability and do not increase computational complexity, that is*

1. *If* $a \in \mathsf{U}\ [\Gamma]\ \mathsf{comp}\ \beta$ *then* $< a >\ \mathsf{type}\ [\Gamma]\ \mathsf{comp}\ \beta' \leq \beta$
2. *If* $a = b \in \mathsf{U}\ [\Gamma]\ \mathsf{comp}\ \beta$ *then* $< a >=< b >\ [\Gamma]\ \mathsf{comp}\ \beta' \leq \beta$

Proof. By induction on the computational complexity β.
Case 1.
Subcase $\Gamma = \emptyset$.

(evaluation) $< a > \Rrightarrow G_{<a>}$ immediately follows from the computability of the judgement $a \in \mathsf{U}$ by using the suitable computation rule.

(correct evaluation) the required derivation can be obtained by applying the suitable U-equality rule to premises whose existence is guaranteed by the computability of the judgement $a \in \mathsf{U}$. For instance, if $a \Rightarrow \pi(a', a'')$ the required proof is the following:

$$\dfrac{\dfrac{a = \pi(a', a'') \in \mathsf{U}}{< a >=< \pi(a', a'') >} \quad \dfrac{a' \in \mathsf{U} \quad a''(x) \in \mathsf{U}\ [x :< a' >]}{< \pi(a', a'') >= \Pi(< a' >, (x) < a''(x) >)}}{< a >= \Pi(< a' >, (x) < a''(x) >)}$$

(parts) the parts of $G_{<a>}$ are computable type(s) with computational complexity less or equal of that one of the corresponding parts of the computable judgement $a \in \mathsf{U}$. For instance, supposing $G_{<a>} \equiv \Pi(< a' >, (x) < a''(x) >)$, by inductive hypothesis, we obtain both that $a' \in \mathsf{U}\ \mathsf{comp}\ \beta'$ implies that $< a' >\ \mathsf{type}\ \mathsf{comp}\ \gamma' \leq \beta'$ and that $a''(x) \in \mathsf{U}\ [x :< a' >]\ \mathsf{comp}\ \beta''$ implies that $< a''(x) >\ \mathsf{type}\ [x :< a' >]\ \mathsf{comp}\ \gamma'' \leq \beta''$. The other cases are completely similar.
Subcase $\Gamma \neq \emptyset$.

(substitution :=) immediately follows by inductive hypothesis (1)

(substitution $\leftarrow$) immediately follows by inductive hypothesis (2)
Case 2.
Subcase $\Gamma = \emptyset$.

(associate) from $a = b \in \mathsf{U}\ \mathsf{comp}\ \beta$ we obtain $a \in \mathsf{U}\ \mathsf{comp}\ \beta_1 < \beta$ and $b \in \mathsf{U}\ \mathsf{comp}\ \beta_2 < \beta$. Hence by inductive hypothesis, point 1,

$$< a >\ \mathsf{type}\ \mathsf{comp}\ \beta_1' \leq \beta_1$$

and

$$< b > \ \mathsf{type\ comp}\ \beta_2' \leq \beta_2$$

(parts) if $< a > \Rrightarrow G_{<a>}$ and $< b > \Rrightarrow G_{<b>}$ then the parts of $G_{<a>}$ and $G_{<b>}$ are equal computable types with the same computational complexity of the corresponding parts of the computable judgement $a = b \in \mathsf{U}$. Let us analyze the case $G_{<a>} \equiv \Pi(< a' >, (x) < a''(x) >)$:

$$\begin{array}{lll} G_{<a>} \equiv \Pi(< a' >, (x) < a''(x) >) & \text{iff} & a \Rightarrow \pi(a', a'') \\ & \text{iff} & b \Rightarrow \pi(b', b'') \\ & \text{iff} & G_{<b>} \equiv \Pi(< b' >, (x) < b''(x) >) \end{array}$$

Moreover, by inductive hypothesis, we obtain both that $a' = b' \in \mathsf{U}\ \mathsf{comp}\ \beta_3$ implies that $< a' >=< b' >\ \mathsf{comp}\ \beta_3' \leq \beta_3$ and that $a''(x) = b''(x) \in \mathsf{U}\ [x :< a' >]\ \mathsf{comp}\ \beta_4$ implies that $< a''(x) >=< b''(x) >\ [x :< a' >]\ \mathsf{comp}\ \beta_4' \leq \beta$.

Hence $\beta' = \bigvee(\beta_1' + 1, \beta_2' + 1, \beta_3' + 1, \beta_4' + 1) \leq \bigvee(\beta_1 + 1, \beta_2 + 1, \beta_3 + 1, \beta_4 + 1) = \beta$. The other cases are completely similar.

Subcase $\Gamma \neq \emptyset$.

(associate) immediately follows by inductive hypothesis, point 1.

(substitution :=) immediately follows by inductive hypothesis, point 2.

(substitution $\leftarrow$) immediately follows by inductive hypothesis, point 2.

Note that the computational complexity of the judgements in the premise and in the conclusion are usually equal, but we may also have different complexities. For instance, the complexity of a basic judgement, like $\mathsf{N\ type}$, is 0 while the complexity of the corresponding judgement, $< n > \in \mathsf{U}$, is 1 due to the requirement that the associate judgement $\mathsf{U\ type}$ is computable.

3.9.3 The structural rules

In the previous section we dealt with the reflexivity on elements and the reflexivity on types rule. All the other structural rules, that is the transitivity on elements, the transitivity on types, the symmetry on elements, the symmetry on types, the equal elements and the equal types rules, are considered together in the next lemma 3.9.3. This lemma is a key point in the proof of computability since it establishes, besides the fact that structural rules preserve computability, other basic and important relationships among the computational complexities of related judgements. As we already pointed out, these information are essential in the subsequent proof since they guarantee the applicability of the inductive hypothesis when we proceed by induction on the computational complexity of a judgement.

Lemma 3.9.3 (Structural rules) *Let β be a computational complexity. Then*

1. *If $a = c \in A\ [\Gamma]\ \mathsf{comp}\ \alpha_1 < \beta$, $b = d \in A\ [\Gamma]\ \mathsf{comp}\ \alpha_2 < \beta$ and $a = b \in A\ [\Gamma]\ \mathsf{comp}\ \alpha$ then*
   - *(i) $\alpha_1 = \alpha_2 = \alpha$*
   - *(ii) $c = d \in A\ [\Gamma]\ \mathsf{comp}\ \alpha$*
   - *(1.1) (transitivity on elements) If $a = b \in A\ [\Gamma]\ \mathsf{comp}\ \alpha_1 < \beta$ and $b = c \in A\ [\Gamma]\ \mathsf{comp}\ \alpha_2 < \beta$ then $a = c \in A\ [\Gamma]\ \mathsf{comp}\ \alpha = \alpha_1 = \alpha_2$*
2. *If $A = C\ [\Gamma]\ \mathsf{comp}\ \alpha_1 < \beta$, $B = D\ [\Gamma]\ \mathsf{comp}\ \alpha_2 < \beta$ and $A = B\ [\Gamma]\ \mathsf{comp}\ \alpha$ then*
   - *(i) $\alpha_1 = \alpha_2 = \alpha$*
   - *(ii) $C = D\ [\Gamma]\ \mathsf{comp}\ \alpha$*
   - *(2.1) (transitivity on types) If $A = B\ [\Gamma]\ \mathsf{comp}\ \alpha_1 < \beta$ and $B = C\ [\Gamma]\ \mathsf{comp}\ \alpha_2 < \beta$ then $A = C\ [\Gamma]\ \mathsf{comp}\ \alpha = \alpha_1 = \alpha_2$*
3. *If $A = C\ [\Gamma]\ \mathsf{comp}\ \beta$ then*
   - *(i.) (associate judgements) $A\ \mathsf{type}\ [\Gamma]\ \mathsf{comp}\ \alpha$ if and only if $C\ \mathsf{type}\ [\Gamma]\ \mathsf{comp}\ \alpha$*
   - *(ii.) (symmetry on types) $C = A\ [\Gamma]\ \mathsf{comp}\ \beta$*

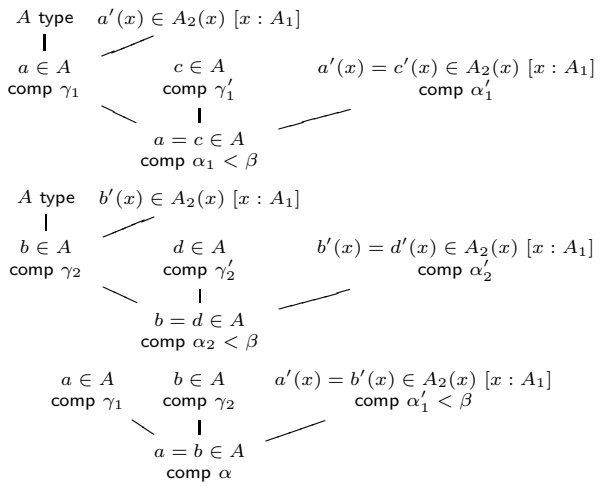


Figure 3.3: Computation trees (Point 1.i. Π-case)

- *(iii.) (element in equal types and equal elements in equal types)*
  - *(iii.a)* $a \in A\ [\Gamma]$ comp α *if and only if* $a \in C\ [\Gamma]$ comp α
  - *(iii.b)* $a = c \in A\ [\Gamma]$ comp α *if and only if* $a = c \in C\ [\Gamma]$ comp α
- *(iv.) (assumption in equal types)* $J\ [\Gamma, x : A]$ comp α *if and only if* $J\ [\Gamma, x : C]$ comp α

4. If $a = c \in A\ [\Gamma]$ comp β *then*

- *(i.) (associate judgements)* $a \in A\ [\Gamma]$ comp α *if and only if* $c \in A\ [\Gamma]$ comp α
- *(ii.) (symmetry on elements)* $c = a \in A\ [\Gamma]$ comp β

Proof. By principal induction on the computational complexity β. The base cases are obvious.

- **Point 1.** The proof follows by subordinate induction on the computational complexity α. As regards point 1.ii, a derivation for the judgement $c = d \in A\ [\Gamma]$ can easily be found by using symmetry and transitivity rules and the derivations for $a = c \in A\ [\Gamma]$, $b = d \in A\ [\Gamma]$ and $a = b \in A\ [\Gamma]$.

  Subcase $\Gamma = \emptyset$. Let G_A be the canonical value of A. The proof varies according to the outermost constant of G_A, but there is a common pattern. First, we prove that the three considered judgements have corresponding part judgements with the same computational complexity. Then, a similar result follows also for the corresponding associate judgements. Here we analyze only three main cases; the other cases are similar.

  - $G_A \equiv \Pi(A_1, A_2)$, and $a \Rightarrow \lambda(a')$, $b \Rightarrow \lambda(b')$, $c \Rightarrow \lambda(c')$ and $d \Rightarrow \lambda(d')$.

    (Point 1.i Π-case) We know that (see fig. 3.3)
    - $a'(x) = c'(x) \in A_2(x)\ [x : A_1]$ comp $\alpha'_1 < \alpha_1 < \beta$
    - $b'(x) = d'(x) \in A_2(x)\ [x : A_1]$ comp $\alpha'_2 < \alpha_2 < \beta$
    - $a'(x) = b'(x) \in A_2(x)\ [x : A_1]$ comp $\alpha' < \alpha$

    hence, by subordinate inductive hypothesis, $\alpha'_1 = \alpha'_2 = \alpha'$. Hence the parts have the same computational complexity. Note that $\alpha' < \beta$.

    As regards to the associate judgements, we obtain by inductive hypothesis 4.i, that:

    - $a = c \in A$ comp $\alpha_1 < \beta$ yields

$$\gamma_1 = \mathsf{comp}(a \in A) = \mathsf{comp}(c \in A) = \gamma'_1$$

* $b = d \in A \ \mathsf{comp}\ \alpha_2 < \beta$ yields

$$\gamma_2 = \mathsf{comp}(b \in A) = \mathsf{comp}(d \in A) = \gamma_2'$$

* $a'(x) = b'(x) \in A_2(x)\ [x : A_1] \ \mathsf{comp}\ \alpha' < \beta$ yields

$$\mathsf{comp}(a'(x) \in A_2(x)\ [x : A_1]) = \mathsf{comp}(b'(x) \in A_2(x)\ [x : A_1])$$

which guarantees $\gamma_1 = \mathsf{comp}(a \in A) = \mathsf{comp}(b \in A) = \gamma_2$. Hence $\gamma_1' = \gamma_1 = \gamma_2 = \gamma_2'$.
(Point 1.ii Π-case) We proved that $\mathsf{comp}(c \in A) = \mathsf{comp}(d \in A)$; hence $c = d \in A \ \mathsf{comp}\ \alpha$ follows by using inductive hypothesis (point 1.ii) on the part judgements of the four considered judgements.

– $G_A \equiv \Sigma(A_1, A_2)$, and $a \Rightarrow \langle a', a'' \rangle$, $b \Rightarrow \langle b', b'' \rangle$, $c \Rightarrow \langle c', c'' \rangle$ and $d \Rightarrow \langle d', d'' \rangle$.
(Point 1.i Σ-case) We know that (see fig. 3.4)
  * $a' = c' \in A_1 \ \mathsf{comp}\ \alpha_1' < \alpha_1 < \beta$
  * $b' = d' \in A_1 \ \mathsf{comp}\ \alpha_2' < \alpha_2 < \beta$
  * (*) $a' = b' \in A_1 \ \mathsf{comp}\ \alpha_1 < \alpha$

  hence, by subordinate inductive hypothesis, $\alpha_1' = \alpha_2' = \alpha' < \beta$. Moreover
  * $a'' = c'' \in A_2(a') \ \mathsf{comp}\ \alpha_1'' < \alpha_1 < \beta$
  * $b'' = d'' \in A_2(b') \ \mathsf{comp}\ \alpha_2'' < \alpha_2 < \beta$
  * $a'' = b'' \in A_2(a') \ \mathsf{comp}\ \alpha'' < \alpha$.

  By using (*), we obtain

$$\begin{array}{lcl} \mathsf{comp}(A_2(a') = A_2(b')) & < & \mathsf{comp}(A_2(x)\ \mathsf{type}\ [x : A_1]) \\ & < & \mathsf{comp}(A\ \mathsf{type}) < \mathsf{comp}(a \in A) \\ & < & \mathsf{comp}(a = c \in A) \\ & < & \beta \end{array}$$

  then, by inductive hypothesis (point 3.iii.b), we have

$$\mathsf{comp}(b'' = d'' \in A2_2(a')) = \alpha_2'' < \beta$$

  hence, by subordinate inductive hypothesis, $\alpha_1'' = \alpha_2'' = \alpha'' < \beta$.

  The proof proceeds analogously to the Π-case. Simply note that to prove that the three considered judgements have corresponding associate judgements with the same computational complexity we need the inductive hypothesis (point 3.iii.a) to obtain that $\mathsf{comp}(b'' \in A_2(a')) = \mathsf{comp}(b'' \in A_2(b'))$.
  (Point 1.ii. Σ-case) Since $A_2(a') = A_2(b') \ \mathsf{comp}\ \delta < \beta$, from $b'' = d'' \in A_2(b') \ \mathsf{comp}\ \alpha_2''$ by inductive hypothesis points 3.ii and 3.iii.b we obtain

$$b'' = d'' \in A_2(a') \ \mathsf{comp}\ \alpha_2''$$

  Then, by using inductive hypothesis point 1.ii, from $a'' = c'' \in A_2(a') \ \mathsf{comp}\ \alpha_1''$, $b'' = d'' \in A_2(a') \ \mathsf{comp}\ \alpha_2''$ and $a'' = b'' \in A_2(a') \ \mathsf{comp}\ \alpha''$ we can obtain that $c'' = d'' \in A_2(a') \ \mathsf{comp}\ \alpha''$.
  From this, since $A_2(a') = A_2(c') \ \mathsf{comp}\ \gamma < \beta$, by inductive. hypothesis. point 3.iii.b, we obtain $c'' = d'' \in A_2(c') \ \mathsf{comp}\ \alpha''$.
  We can easily prove also that $c \in A \ \mathsf{comp}\ \gamma_1$, $d \in A \ \mathsf{comp}\ \gamma_2$, $c' = d' \in A \ \mathsf{comp}\ \alpha'$, and hence that $c = d \in A \ \mathsf{comp}\ \alpha$.

– $G_A \equiv \mathsf{U}$.
  We will develop in a detailed way only the case $a \Rightarrow \pi(a', a'')$, $b \Rightarrow \pi(b', b'')$, $c \Rightarrow \pi(c', c'')$ and $d \Rightarrow \pi(d', d'')$.
  (Point 1.i U-case) We know that

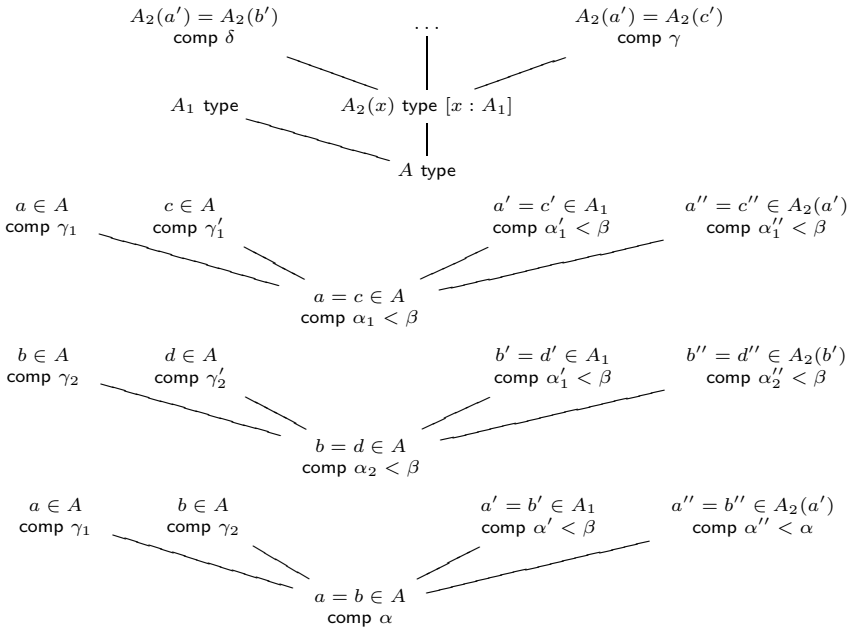


Figure 3.4: Computation trees (Point 1.i. Σ-case)

- $a' = c' \in \mathsf{U}$ comp $\alpha_1' < \alpha_1 < \beta$
- $b' = d' \in \mathsf{U}$ comp $\alpha_2' < \alpha_2 < \beta$
- $a' = b' \in \mathsf{U}$ comp $\alpha' < \alpha$

hence, by subordinate inductive hypothesis, $\alpha_1' = \alpha_2' = \alpha' < \beta$. Moreover we know that

- $a''(x) = c''(x) \in \mathsf{U}\ [x :< a' >]$ comp $\alpha_1'' < \alpha_1 < \beta$
- $b''(x) = d''(x) \in \mathsf{U}\ [x :< b' >]$ comp $\alpha_2'' < \alpha_2 < \beta$
- $a''(x) = b''(x) \in \mathsf{U}\ [x :< a' >]$ comp $\alpha'' < \alpha$.

Since $a' = b' \in \mathsf{U}$ comp $\alpha' < \beta$, we obtain $< a' >=< b' >$ comp $\gamma' \leq \alpha' < \beta$, by lemma 3.9.2, and then, by inductive hypothesis (3.iv), $b''(x) = d''(x) \in \mathsf{U}\ [x :< a' >]$ comp α_2'' hence, by subordinate inductive hypothesis, $\alpha_1'' = \alpha_2'' = \alpha'' < \beta$. Analogously to the previous cases, it is now possible to prove that the three considered judgements have corresponding associate judgements of the same computational complexity .

(Point 1.ii. U-case) The proof proceeds analogously to the previous cases. It is worth to describe only the proof that

$$c''(x) = d''(x) \in \mathsf{U}\ [x :< c' >] \text{ comp } \alpha''$$

By subordinate inductive hypothesis we know that

$$c''(x) = d''(x) \in \mathsf{U}\ [x :< a' >] \text{ comp } \alpha''$$

Since $a' = c' \in \mathsf{U}$ comp $\alpha_1' < \beta$, by lemma 3.9.2, we obtain

$$< a' >=< c' > \text{ comp } \gamma_1' \leq \alpha_1' < \beta$$

and then, by inductive hypothesis (point 3.iv), we obtain the thesis.

Subcase $\Gamma \neq \emptyset$

(Point 1.i) We prove that the three considered judgements have the corresponding associate judgements and substituted judgements of the same computational complexity.

(substitution :=) The result is immediate by subordinate inductive hypothesis (see fig. 3.5).

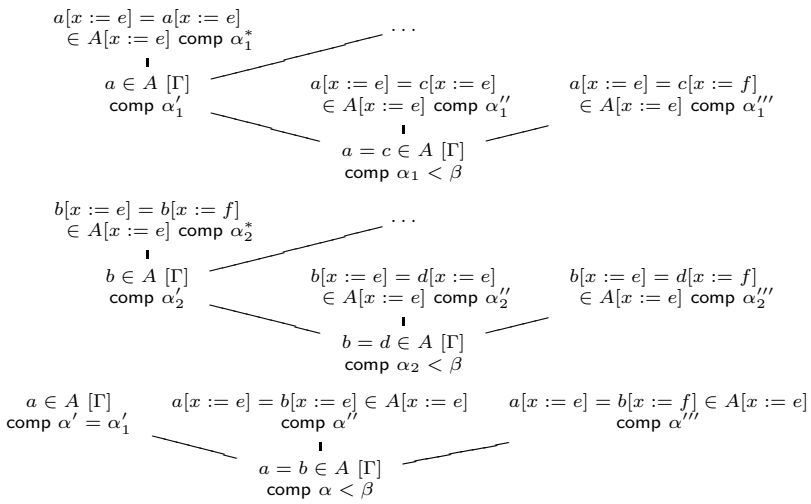


Figure 3.5: Point 1.i (substitution :=)

(substitution ←) First observe that, by subordinate inductive hypothesis (1.i) $\alpha'' = \alpha_1''' = \alpha_2'''$. Moreover, if $e = f$: is a c.c.s. fitting with Γ this holds also for e :. Hence $\alpha_1^* < \beta$ and $\alpha_2^* < \beta$ and, by subordinate inductive hypothesis (points 1.i and 1.ii), $\alpha'' = \alpha_1^* = \alpha_2^* = \alpha'''$. Hence $\alpha''' = \alpha_1''' = \alpha_2'''$.

(associate judgements) Since $\alpha' = \alpha_1'$, $\alpha'' = \alpha_1''$ and $\alpha''' = \alpha_1'''$ then $\alpha = \alpha_1 < \beta$ and, by inductive hypothesis 4.i, $\mathsf{comp}(b \in A\ [\Gamma]) = \alpha_2' = \mathsf{comp}(a \in A\ [\Gamma]) = \alpha_1'$.

(Point 1.ii)

By inductive hypothesis (point 4.i) $\mathsf{comp}(c \in A\ [\Gamma]) = \mathsf{comp}(a \in A\ [\Gamma]) = \alpha'$. Then, by subordinate inductive hypothesis on substituted judgements, we obtain that, for any c.c.s. e :, $\mathsf{comp}(c[x := e] = d[x := e] \in A[x := e]) = \alpha''$ holds and for any c.c.s. $e = f$:, $\mathsf{comp}(c[x := e] = d[x := f] \in A[x := e]) = \alpha''$ holds. Hence $\mathsf{comp}(c = d \in A\ [\Gamma]) = \alpha$.

Point 1.1

By inductive hypothesis (point 4.ii), from $\mathsf{comp}(a = b \in A\ [\Gamma]) = \alpha_1 < \beta$ we obtain that $\mathsf{comp}(b = a \in A\ [\Gamma]) = \alpha_1 < \beta$. Hence $b \in A\ [\Gamma]$ is computable and, by the reflexivity lemma 3.8.5, $b = b \in A\ [\Gamma]$ is computable. Then the result follows from the previous point 1.ii.

- **Point 2.**

  The proof follows by subordinate induction on the computational complexity α. The proof of this case is analogous to the one of point 1. Just substitute the judgement 'equal elements in a type' with the judgement 'equal types' and pay attention in analyzing the case $G_A \equiv \mathsf{Eq}(A_1, e, f)$ where an obvious application of point 1. is required.

  Point 2.1.

  By using inductive hypothesis point 3.ii, from $\mathsf{comp}(A = B\ [\Gamma]) = \alpha_1 < \beta$ we obtain that $\mathsf{comp}(B = A\ [\Gamma]) = \alpha_1 < \beta$. Hence B type $[\Gamma]$ is computable and, by the reflexivity lemma 3.8.6, $B = B\ [\Gamma]$ is computable. Then the result follows from the previous point 2.ii.

- **Point 3.**

  Point 3.i Subcase $\Gamma = \emptyset$.

  Let $A \Rrightarrow G_A$ and $C \Rrightarrow G_C$ we must prove that the parts of G_A and G_C have the same computational complexity. The proof varies according to the outermost constant of G_A. We analyze only two significant cases; the other cases are similar.

– $G_A \equiv \Pi(A_1, A_2)$ hence, since $A = C$ is a computable judgement, $G_C \equiv \Pi(C_1, C_2)$ and we know that

$$(*) \qquad \mathsf{comp}(A_1 = C_1) < \beta$$

and hence, by inductive hypothesis point 3.i, $\mathsf{comp}(A_1\ \mathsf{type}) = \mathsf{comp}(C_1\ \mathsf{type})$;

$$(**) \qquad \mathsf{comp}(A_2(x) = C_2(x)\ [x : A_1]) < \beta$$

and hence $\mathsf{comp}(A_2(x)\ \mathsf{type}\ [x : A_1]) = \mathsf{comp}(C_2(x)\ \mathsf{type}\ [x : A_1])$, by inductive hypothesis point 3.i, and finally $\mathsf{comp}(A_2(x)\ \mathsf{type}\ [x : A_1]) = \mathsf{comp}(C_2(x)\ \mathsf{type}\ [x : C_1])$ by using inductive hypothesis point 3.iv.

– $G_A \equiv \mathsf{Eq}(A_1, a', a'')$ and $G_C \equiv \mathsf{Eq}(C_1, c', c'')$ and we know that

$$(*) \qquad \mathsf{comp}(A_1 = C_1) < \beta$$

then, by inductive hypothesis point 3.i, $\mathsf{comp}(A_1\ \mathsf{type}) = \mathsf{comp}(C_1\ \mathsf{type})$;

$$(**) \qquad \mathsf{comp}(a' = c' \in A_1) < \beta$$

hence, by inductive hypothesis point 4.i, $\mathsf{comp}(a' \in A_1) = \mathsf{comp}(c' \in A_1)$ and then, by inductive hypothesis point 3.iii.a, $\mathsf{comp}(a' \in A_1) = \mathsf{comp}(c' \in C_1)$;

$$(***) \qquad \mathsf{comp}(a'' = c'' \in A_1) < \beta$$

hence, by inductive hypothesis point 4.i, $\mathsf{comp}(a'' \in A_1) = \mathsf{comp}(c'' \in A_1)$ and then, by inductive hypothesis point 3.iii.a, $\mathsf{comp}(a'' \in A_1) = \mathsf{comp}(c'' \in C_1)$.

Point 3.i. Subcase $\Gamma \neq \emptyset$

For any c.c.s. e :, since $\mathsf{comp}(A[x := e] = C[x := e]) < \beta$, by inductive hypothesis 3.i, it immediately follows that

$$\mathsf{comp}(A[x := e]\ \mathsf{type}) = \mathsf{comp}(C[x := e]\ \mathsf{type})$$

For any c.c.s. $e = f$: we know that e : and, by reflexivity, $e = e$: are c.c.s fitting with Γ. Hence,

$$\mathsf{comp}(A[x := e] = A[x := e]) = \mathsf{comp}(A[x := e] = A[x := f])$$

by inductive hypothesis 2.i, and

$$\mathsf{comp}(A[x := e] = A[x := e]) = \mathsf{comp}(C[x := e] = C[x := f])$$

by inductive hypothesis 2.ii. Hence

$$\mathsf{comp}(A[x := e] = A[x := f]) = \mathsf{comp}(C[x := e] = C[x := f])$$

and thus $\mathsf{comp}(A\ \mathsf{type}\ [\Gamma]) = \mathsf{comp}(C\ \mathsf{type}\ [\Gamma])$

Point 3.ii

(associate judgements) We must prove that the associate judgements of the derivable judgement $C = A\ [\Gamma]$ are computable and their computational complexities are equal to the computational complexities of the corresponding associate of the judgement $A = C\ [\Gamma]$. This result is obvious by the previous point 3.i.

Point 3.ii. Subcase $\Gamma = \emptyset$.

We must prove that the parts of $C = A$ have the same computational complexity of the parts of $A = C$. Let us show only two cases

– $G_A \equiv \Pi(A_1, A_2)$ and hence $G_C \equiv \Pi(C_1, C_2)$.
The parts of $A = C$ are $A_1 = C_1\ \mathsf{comp}\ \alpha_1$ and $A_2(x) = C_2(x)\ [x : A_1]\ \mathsf{comp}\ \alpha_2$ and the parts of $C = A$ are $C_1 = A_1\ \mathsf{comp}\ \alpha_1'$ and $C_2(x) = A_2(x)\ [x : C_1]\ \mathsf{comp}\ \alpha_2'$. Now $\alpha_1 = \alpha_1'$, by inductive hypothesis point 3.ii, because $\alpha_1 < \beta$ and, since, by inductive hypothesis point 3.v,

$$\mathsf{comp}(C_2(x) = A_2(x)\ [x : A_1]) = \alpha_2'$$

it follows that $\alpha_2 = \alpha_2'$, again by using the inductive hypothesis point 3.ii, because $\alpha_2 < \beta$.

– $G_A \equiv \mathsf{Eq}(A_1, a', a'')$ and hence $G_C \equiv \mathsf{Eq}(C_1, c', c'')$.
The parts of the judgement $A = C$ are $A_1 = C_1$ comp α_1, $a' = c' \in A_1$ comp α_2 and $a'' = c'' \in A_1$ comp α_3 while the parts of $C = A$ are $C_1 = A_1$ comp α_1', $c' = a' \in C_1$ comp α_2' and $c'' = a'' \in C_1$ comp α_3'. Now $\alpha_1 = \alpha_1'$ by inductive hypothesis point 3.ii because $\alpha_1 < \beta$. Moreover, since, by inductive hypothesis point 4.ii, both $\mathsf{comp}(c' = a' \in A_1) = \alpha_2$ and $\mathsf{comp}(c'' = a'' \in A_1) = \alpha_3$, we obtain $\mathsf{comp}(c' = a' \in C_1) = \alpha_2$ and $\mathsf{comp}(c'' = a'' \in C_1) = \alpha_3$ by using inductive hypothesis point 3.iv.

Point 3.ii. Subcase $\Gamma \neq \emptyset$.

We have to prove that the two considered judgements have substituted judgements with the same computational complexity.

(substitution :=) Immediate by inductive hypothesis 3.ii.

(substitution $\leftarrow$) For any c.c.s. $e = f$: fitting with Γ also e : is a c.c.s. fitting with Γ. Hence by inductive hypothesis (2.i and 2.ii)

$$\begin{array}{rcl} \mathsf{comp}(A[x := e] = C[x := e]) & = & \mathsf{comp}(A[x := e] = A[x := f]) \\ & = & \mathsf{comp}(A[x := e] = A[x := e]) \\ & = & \mathsf{comp}(C[x := e] = A[x := f]) \end{array}$$

$$\begin{array}{rcl} \mathsf{comp}(A[x := e] = C[x := e]) & = & \mathsf{comp}(A[x := e] = C[x := f]) \\ & = & \mathsf{comp}(A[x := e] = A[x := e]) \\ & = & \mathsf{comp}(C[x := e] = C[x := f]) \end{array}$$

Hence $\mathsf{comp}(C[x := e] = A[x := f]) = \mathsf{comp}(A[x := e] = C[x := f])$.

Point 3.iii.a.

Let us prove the if-part (the proof of the only-if part is completely similar) by subordinate induction on the computational complexity α. Note that, by point 3.i, the associate judgements of $a \in A$ [Γ] and $a \in C$ [Γ] are computable judgements with the same complexity.

Point 3.iii.a. Subcase $\Gamma = \emptyset$.

(evaluation) $a \Rightarrow g_a$, by hypothesis

(correct evaluation) $a = g_a \in C$ is provable, immediate

(parts) Let us analyze here only three cases according to the form of G_C.

– $G_C \equiv \Sigma(C_1, C_2)$ and hence $G_A \equiv \Sigma(A_1, A_2)$ and $g_a \equiv \langle a', a'' \rangle$ and we know that
  * $A_1 = C_1$ comp $\alpha_1 < \beta$,
  * $A_2(x) = C_2(x)$ $[x : A_1]$ comp $\alpha_2 < \beta$,
  * $a' \in A_1$ comp $\alpha' < \alpha$ and
  * $a'' \in A_2(a')$ comp $\alpha'' < \alpha$

  and hence $\mathsf{comp}(a' \in C_1) = \alpha'$, by inductive hypothesis point 3.iii.a, and, since

$$\begin{array}{rcl} \mathsf{comp}(A_2(a') = C_2(a')) & < & \mathsf{comp}(A_2(x) = C_2(x)\ [x : A_1]) \\ & < & \mathsf{comp}(A = C) \\ & < & \beta \end{array}$$

  we obtain

$$a'' \in C_2(a') \text{ comp } \alpha''$$

  by using inductive hypothesis point 3.iii.a.

– $G_C \in \mathsf{Eq}(C_1, c', c'')$ and hence $G_A \equiv \mathsf{Eq}(A_1, a', a'')$ and $g_a \equiv \mathsf{e}$ and we know that
  * $a' = c' \in A_1$ comp $\alpha_2 < \beta$,
  * $a'' = c'' \in A_1$ comp $\alpha_3 < \beta$ and
  * $a' = a'' \in A_1$ comp $\alpha' < \alpha$

and, by point 1., we obtain $c' = c'' \in A_1 \ \mathsf{comp}\ \alpha'$ then, since $A_1 = C_1 \ \mathsf{comp}\ \alpha_1 < \beta$, by using inductive hypothesis point 3.iii.b,

$$\mathsf{comp}(c' = c'' \in C_1) = \alpha'$$

– $G_C \equiv \mathsf{W}(C_1, C_2)$ and hence $G_A \equiv \mathsf{W}(A_1, A_2)$ and $g_a \equiv \mathsf{sup}(a', a'')$ and we know that $A_1 = C_1 \ \mathsf{comp}\ \alpha_1 < \beta$ and $a' \in A_1 \ \mathsf{comp}\ \alpha' < \alpha$; then, by inductive hypothesis 3.iii.a, $a' \in C_1 \ \mathsf{comp}\ \alpha'$. Moreover we know that $a''(y) \in \mathsf{W}(A_1, A_2)\ [y : A_2(a')] \ \mathsf{comp}\ \alpha'' < \alpha$ and $A_2(x) = C_2(x)\ [x : A_1] \ \mathsf{comp}\ \alpha_2 < \beta$.

Finally also $a''(y) \in \mathsf{W}(A_1, A_2)\ [y : C_2(a')] \ \mathsf{comp}\ \alpha'' < \alpha$ holds, by inductive hypothesis point 3.iv, since $\mathsf{comp}(A_2(a') = C_2(a')) < \mathsf{comp}(A_2(x) = C_2(x)\ [x : A_1]) < \beta$. Now to prove

$$a''(y) \in \mathsf{W}(C_1, C_2)\ [y : C_2(a')] \ \mathsf{comp}\ \alpha''$$

we must consider the associate judgements. Since

$$\mathsf{comp}(\mathsf{W}(A_1, A_2)\ \mathsf{type}) = \mathsf{comp}(\mathsf{W}(C_1, C_2)\ \mathsf{type})$$

and

$$\mathsf{comp}(\mathsf{W}(A_1, A_2) = \mathsf{W}(A_1, A_2)) = \mathsf{comp}(\mathsf{W}(C_1, C_2) = \mathsf{W}(C_1, C_2))$$

then

$$\mathsf{comp}(\mathsf{W}(A_1, A_2)\ \mathsf{type}\ [y : C_2(a')]) = \mathsf{comp}(\mathsf{W}(C_1, C_2)\ \mathsf{type}\ [y : C_2(a')])$$

(substitutions) By subordinate inductive hypothesis 3.iii.a and 3.iii.b, for any c.c.s. $e \in C_2(a')$ and $e = f \in C_2(a')$ we have

$$\mathsf{comp}(a''(e) \in \mathsf{W}(A_1, A_2)) = \mathsf{comp}(a''(e) \in \mathsf{W}(C_1, C_2))$$

and

$$\mathsf{comp}(a''(e) = a''(f) \in \mathsf{W}(A_1, A_2)) = \mathsf{comp}(a''(e) = a''(f) \in \mathsf{W}(C_1, C_2))$$

Point 3.iii.a. Subcase $\Gamma \neq \emptyset$.

(substitution :=) The result immediately follows by using the inductive hypothesis point 3.iii.a.

(substitution $\leftarrow$) The result immediately follows by using the inductive hypothesis point 3.iii.b.

Point 3.iii.b.

The proof of this case is similar to the previous point 3.iii.a. Just note that, by point 3.iii.a, the associate judgements of $a = c \in A\ [\Gamma]$ and $c = a \in C\ [\Gamma]$ are computable with the same complexity.

Point 3.iv.

The proof is by subordinate induction on the computational complexity α. Let us prove the if-part (the proof of the only if-part is completely similar).

(associate judgements) By subordinate inductive hypothesis the associate judgements of $J\ [\Gamma, x : C]$ are computable with the same complexity of those of $J\ [\Gamma, x : A]$.

(substitutions) In order to analyse the substituted judgements, suppose Γ is the context $[y_1 : B_1, \ldots, y_n : B_n]$. Since $A = C\ [\Gamma] \ \mathsf{comp}\ \beta$ then for any c.c.s. $a_1 : B_1,\ \ldots,\ a_n : B_n$, $e : C$ (or $a_1 = a_1' : B_1,\ \ldots,\ a_n = a_n' : B_n,\ e = e' : C$) fitting with $[\Gamma, x : C]$, we have $\mathsf{comp}(A[y := a] = C[y := a]) < \beta$ hence, by point 3.iii.a (or 3.iii.b), $e : A$ (or $e = e' : A$) is a computable judgement and thus the same substitution fits also with $[\Gamma, x : A]$. Hence the substituted judgements are computable with the same complexity.

- **Point 4.**

  Point 4.i.

  Let us prove the if-part (the proof of the only if-part is completely similar).

  (associate judgements) The associate judgement of both judgements is A type.

  Point 4.i. Subcase $\Gamma = \emptyset$.

  We must prove only that the parts of $c \in A$ have the same computational complexity of the corresponding parts of $a \in A$. According to the values of A and a we show here only three cases:

  - $A \Rrightarrow \Sigma(A_1, A_2)$ and $a \Rightarrow \langle a', a'' \rangle$.
    We know that $c \Rightarrow \langle c', c'' \rangle$, $\mathsf{comp}(a' = c' \in A_1) < \beta$ and $\mathsf{comp}(a'' = c'' \in A_2(a')) < \beta$; therefore $\mathsf{comp}(a' \in A_1) = \mathsf{comp}(c' \in A_1)$ by inductive hypothesis 4.i; and, since

    $$\begin{array}{rcl} \mathsf{comp}(A_2(a') = A_2(c')) & < & \mathsf{comp}(A_2(x) \text{ type } [x : A_1]) \\ & < & \mathsf{comp}(A \text{ type}) < \mathsf{comp}(a \in A) \\ & < & \mathsf{comp}(a = c \in A) \\ & = & \beta \end{array}$$

    we obtain

    $$\mathsf{comp}(a'' \in A_2(a')) = \mathsf{comp}(c'' \in A_2(c'))$$

    by using inductive hypothesis point 4.i and point 3.iii.a.
  - $A \Rrightarrow \mathsf{W}(A_1, A_2)$ and $a \Rightarrow \mathsf{sup}(a', a'')$.
    We know that $c \Rightarrow \mathsf{sup}(c', c'')$, and that $\mathsf{comp}(a' = c' \in A_1) < \beta$; therefore

    $$\mathsf{comp}(a' \in A_1) = \mathsf{comp}(c' \in A_1)$$

    by using inductive hypothesis point 4.i. We know also that

    $$\mathsf{comp}(a''(x) = c''(x) \in \mathsf{W}(A_1, A_2) \ [x : A_2(a')]) < \beta$$

    and

    $$\begin{array}{rcl} \mathsf{comp}(A_2(a') = A_2(c')) & < & \mathsf{comp}(A_2(x) = A_2(x) \ [x : A_1]) \\ & < & \mathsf{comp}(A_2(x) \text{ type } [x : A_1] \\ & < & \beta \end{array}$$

    by inductive hypothesis point 4.i and point 3.iv, then we obtain

    $$\begin{array}{rcl} \mathsf{comp}(a''(x) \in \mathsf{W}(A_1, A_2) \ [x : A_2(a')]) & = & \mathsf{comp}(c''(x) \in \mathsf{W}(A_1, A_2) \ [x : A_2(a')]) \\ & = & \mathsf{comp}(c''(x) \in \mathsf{W}(A_1, A_2) \ [x : A_2(c')]) \end{array}$$

  - $A \Rrightarrow \mathsf{U}$ and $a \Rightarrow \pi(a', a'')$, and therefore $c \Rightarrow \pi(c', c'')$.
    We know both that $\mathsf{comp}(a' = c' \in \mathsf{U}) < \beta$ and $\mathsf{comp}(a''(x) = c''(x) \in \mathsf{U}[x :< a' >]) < \beta$. Therefore, by inductive hypothesis (4.i), we obtain both $\mathsf{comp}(a' \in \mathsf{U}) = \mathsf{comp}(c' \in \mathsf{U})$ and $\mathsf{comp}(a''(x) \in \mathsf{U} \ [x :< a' >]) = \mathsf{comp}(c''(x) \in \mathsf{U} \ [x :< a' >])$.
    Moreover, $\mathsf{comp}(a' = c' \in \mathsf{U}) < \beta$. Then, by using lemma 3.9.2, we obtain that $\mathsf{comp}(< a' >=< c' >) < \beta$, and we obtain, by using inductive hypothesis (3.iv),

    $$\mathsf{comp}(c''(x) \in \mathsf{U} \ [x :< a' >]) = \mathsf{comp}(c''(x) \in \mathsf{U} \ [x :< c' >])$$

  Point 4.i. Subcase $\Gamma \neq \emptyset$.

  (substitution $:=$) The result immediately follows by inductive hypothesis 4.i.

  (substitution $\leftarrow$) The result follows by inductive hypothesis point 1 using the fact that if $e = f$: is a c.c.s fitting with Γ so is e :.

  Point 4.ii.

  Point 4.ii. Subcase $\Gamma = \emptyset$.

Since the associate judgement of $a = c \in A$ are exactly those of $c = a \in A$, we must prove only that the parts of $c = a \in A$ have the same computational complexity of the corresponding parts of $a = c \in A$. The proof is similar to that of the previous point 4.i; let us analyze just one case:

- $A \Rrightarrow \Sigma(A_1, A_2)$ and $a \Rightarrow \langle a', a'' \rangle$. We know that $c \Rightarrow \langle c', c'' \rangle$, $\mathsf{comp}(a' = c' \in A_1) < \beta$ and $\mathsf{comp}(a'' = c'' \in A_2(a')) < \beta$; therefore $\mathsf{comp}(a' = c' \in A_1) = \mathsf{comp}(c' = a' \in A_1)$, by inductive hypothesis 4.ii; and, since

$$\begin{array}{rcl} \mathsf{comp}(A_2(a') = A_2(c')) & < & \mathsf{comp}(A_2(x) \ \mathsf{type}\ [x : A_1]) \\ & < & \mathsf{comp}(A \ \mathsf{type}) \\ & < & \mathsf{comp}(a \in A) \\ & < & \mathsf{comp}(a = c \in A) \\ & = & \beta \end{array}$$

we obtain

$$\mathsf{comp}(a'' = c'' \in A_2(a')) = \mathsf{comp}(c'' = a'' \in A_2(c'))$$

by using inductive hypothesis point 4.ii and point 3.iii.b.

Point 4.ii. Subcase $\Gamma \neq \emptyset$.

As for the previous case the result follows by inductive hypothesis (4.i. and 4.ii) by using inductive hypothesis (point 1.1) and the fact that if $e = f$: is a c.c.s fitting with Γ so is e :.

It is worth noting that, the computability of the associate judgements which were left out from the definition 3.7.1 of computable judgement (points 2.2.1 and 2.4.1) is now established by the symmetry rules.

3.9.4 The assumption rules

In section 3.3 we presented the assumption rules of our system; they introduce variables of any arity. The new assumption appears in the context Γ of the conclusion of the rule and for this reason only the case $\Gamma \neq \emptyset$ has to be considered in order to prove that assumption rules preserve computability.

Lemma 3.9.4 (First assumption rule) *Let $a_i : A_i\ [\Gamma_i]$, for any $1 \leq i \leq n$, be n computable judgements and $: B\ [\Gamma] \equiv A(x_1, \ldots, x_n)\ \mathsf{type}\ [\Gamma, x_1 : A_1, \ldots, x_n : A_n]$ be a computable judgement. Then the judgement*

$$y(a_1, \ldots, a_n) \in A(a_1, \ldots, a_n)\ [\Gamma']$$

where Γ' is the merge without duplication of the contexts Γ_1, ..., Γ_n, Γ, $[y : B]$, is computable.

Proof. Let $\Gamma' \equiv s_1 : S_1, \ldots, s_k : S_k, y : B, z_1 : C_1, \ldots, z_m : C_m$, for $k \geq 0$ and $m \geq 0$, where $z_1 : C_1, \ldots, z_m : C_m$ strictly depend on $y : B$. First note that, since the context Γ' extends both the contexts Γ and all the contexts Γ_i, for $1 \leq i \leq n$, by the weakening lemma, we have that

$$A(x_1, \ldots, x_n)\ \mathsf{type}\ [\Gamma', x_1 : A_1, \ldots, x_n : A_n]$$

and

$$a_i : A_i\ [\Gamma'] \qquad 1 \leq i \leq n$$

are computable judgements.

(associate judgement) The computability of the judgement

$$A(a_1, \ldots, a_n)\ \mathsf{type}\ [\Gamma']$$

follows by the substitution lemma.

(substitution :=) Consider any c.c.s. fitting with the context $\Gamma' : d_1 : S_1, \ldots, d_k : S_k, b : B, c_1 : C_1, \ldots, c_m : C_m$. Note that, if $A'_i \equiv ((s_1, \ldots, s_k)\ A_i)(d_1, \ldots, d_k)$ then $b : B$ abbreviates:

$$b : B[s_1 := d_1, \ldots, s_k := d_k] \equiv b(x_1, \ldots, x_n) \in ((s_1, \ldots, s_k)\ A)(d_1, \ldots, d_k)\ [x_1 : A'_1, \ldots, x_n : A'_n]$$

Moreover, for any $1 \leq i \leq n$, $a_i : A_i\ [\Gamma']$ is a computable judgement; then, by the head substitution lemma 3.8.8,

$$a_i : A_i\ [\Gamma'][s_1 := d_1, \ldots, s_k := d_k, y := b, z_1 := c_1, \ldots, z_m := c_m]$$

is also computable, and

$$\begin{array}{ll} & ((s_1, \ldots, s_k, y, z_1, \ldots, z_m)\ A_i)(d_1, \ldots, d_k, b, c_1, \ldots, c_m) \\ \equiv & ((s_1, \ldots, s_k)\ A_i)(d_1, \ldots, d_k) \\ \equiv & A'_i \end{array}$$

Let, for any $1 \leq i \leq n$, $e_i \equiv ((s_1, \ldots, s_k, y, z_1, \ldots, z_m)\ a_i)(d_1, \ldots, d_k, b, c_1, \ldots, c_m)$ then also the judgement $b : B[x_1 := e_1, \ldots, x_n := e_n]$ is computable.

But $y : B$, $z_1 : C_1$, $\ldots$, $z_m : C_m$ cannot appear in the context $[\Gamma, x_1 : A_1, \ldots, x_n : A_n]$; then

$$((s_1, \ldots, s_k, y, z_1, \ldots, z_m)\ A)(d_1, \ldots, d_k, b, c_1, \ldots, c_m) \equiv ((s_1, \ldots, s_k)\ A)(d_1, \ldots, d_k)$$

and therefore

$$\begin{array}{ll} \equiv & b : B[x_1 := e_1, \ldots, x_n := e_n] \\ \equiv & y(a_1, \ldots, a_n) \in A(a_1, \ldots, a_n)\ [\Gamma'][s_1 := d_1, \ldots, s_k := d_k, y := b, z_1 := c_1, \ldots, z_m := c_m] \end{array}$$

(substitution $\leftarrow$) Consider any c.c.s. $d_1 = d'_1 : S_1$, $\ldots$, $d_k = d'_k : S_k$, $b = b' : B$, $c_1 = c'_1 : C_1$, $\ldots$, $c_m = c'_m : C_m$ fitting with Γ'. The, the proof proceeds as before after noting that, for any $1 \leq i \leq n$, the judgement

$$a_i : A_i\ [\Gamma'][s_1 \leftarrow d_1 = d'_1, \ldots, s_k \leftarrow d_k = d'_k, y \leftarrow b = b', z_1 \leftarrow c_1 = c'_1, \ldots, z_m \leftarrow c_m = c'_m]$$

is computable and can be substituted for x_i in $b = b' : B$ obtaining a computable judgement which is exactly

$$\begin{array}{l} y(a_1, \ldots, a_n) \in A(a_1, \ldots, a_n)\ [\Gamma'][s_1 \leftarrow d_1 = d'_1, \ldots, s_k \leftarrow d_k = d'_k, \\ \qquad\qquad y \leftarrow b = b', z_1 \leftarrow c_1 = c'_1, \ldots, z_m \leftarrow c_m = c'_m] \end{array}$$

Lemma 3.9.5 (Second assumption rule) *Let $a_i = a'_i : A_i\ [\Gamma_i]$ for any $1 \leq i \leq n$, be n computable judgements and $: B\ [\Gamma] \equiv A(x_1, \ldots, x_n)$ type $[\Gamma, x_1 : A_1, \ldots, x_n : A_n]$ be a computable judgement. Then the judgement*

$$y(a_1, \ldots, a_n) = y(a'_1, \ldots, a'_n) \in A(a_1, \ldots, a_n)\ [\Gamma']$$

where Γ' is the merge without duplication of the contexts Γ_1, $\ldots$, Γ_n, Γ, $[y : B]$ is computable.

Proof. Let $\Gamma' \equiv s_1 : S_1, \ldots, s_k : S_k, y : B, z_1 : C_1, \ldots, z_m : C_m$, for $k \geq 0$, $m \geq 0$, where $z_1 : C_1, \ldots, z_m : C_m$ strictly depend on $y : B$.

(associate judgement) By hypothesis, for any $1 \leq i \leq n$, $a_i = a'_i : A_i\ [\Gamma_i]$ is a computable judgement; then also its associate judgements $a_i : A_i\ [\Gamma_i]$ is computable. Hence, by the previous lemma, the judgement

$$y(a_1, \ldots, a_n) \in A(a_1, \ldots, a_n)\ [\Gamma']$$

is computable.

(substitution $:=$) Consider any c.c.s. $d_1 : S_1$, $\ldots$, $d_k : S_k$, $b : B$, $c_1 : C_1$, $\ldots$, $c_m : C_m$ fitting with Γ'. The proof proceeds as before after noting that $b = b : B$ is computable by the reflexivity lemma 3.8.5 and that, for any $1 \leq i \leq n$,

$$a_i = a'_i : A_i\ [\Gamma'][s_1 := d_1, \ldots, s_k := d_k, y := b, z_1 := c_1, \ldots, z_m := c_m]$$

is a computable judgement which can be substituted for x_i in $b = b : B$ obtaining a computable judgement which is exactly

$$\begin{array}{l} y(a_1, \ldots, a_n) = y(a'_1, \ldots, a'_n) \in A(a_1, \ldots, a_n)\ [\Gamma'] \\ \qquad\qquad [s_1 := d_1, \ldots, s_k := d_k, y := b, z_1 := c_1, \ldots, z_m := c_m] \end{array}$$

(substitution $\leftarrow$) Consider any c.c.s. $d_1 = d'_1 : S_1$, ..., $d_k = d'_k : S_k$, $b = b' : B$, $c_1 = c'_1 : C_1$, ..., $c_m = c'_m : C_m$ fitting with Γ'. The proof proceeds as before by noting that, for any $1 \leq i \leq n$,

$$a_i = a'_i : A_i \ [\Gamma'][s_1 \leftarrow d_1 = d'_1, \ldots, s_k \leftarrow d_k = d'_k, y \leftarrow b = b', z_1 \leftarrow c_1 = c'_1, \ldots, z_m \leftarrow c_m = c'_m]$$

are computable judgements which can be substituted for x_i in $b = b' : B$ obtaining a computable judgement which is exactly

$$\begin{aligned} y(a_1, \ldots, a_n) = y(a'_1, \ldots, a'_n) \in A(a_1, \ldots, a_n) \ [\Gamma'][s_1 \leftarrow d_1 = d'_1, \ldots, \\ \ldots, s_k \leftarrow d_k = d'_k, y \leftarrow b = b', z_1 \leftarrow c_1 = c'_1, \ldots, z_m \leftarrow c_m = c'_m] \end{aligned}$$

3.9.5 The logical rules

We have now to analyze the rules that we call "logical" since they can be used to interpret a logical intuitionistic first order calculus or a logical theory of natural numbers. An informal discussion on the computability of these rules is usually depicted in many of the descriptions of the intuitionistic type theory and we follow the same ideas. Nevertheless, we want to note that in our experience a complete formal proof of computability for these rules cannot be carried on without a substantial use of lemma 3.9.3 on structural rules.

Lemma 3.9.6 (Π-formation rules) *The Π-formation rules preserve computability. That is*

1. *If $J_1 \equiv A$ type $[\Gamma]$ and $J_2 \equiv B(x)$ type $[\Gamma, x : A]$ are computable judgements then the judgement*
$$\Pi(A, B) \text{ type } [\Gamma]$$
is computable

2. *If $J_1 \equiv A = C \ [\Gamma]$ and $J_2 \equiv B(x) = D(x) \ [\Gamma, x : A]$ are computable judgements then the judgement*
$$\Pi(A, B) = \Pi(C, D) \ [\Gamma]$$
is computable.

Proof By induction on the computational complexity α of J_1.
Case 1.
Subcase $\Gamma = \emptyset$.

(evaluation) $\Pi(A, B) \Rrightarrow \Pi(A, B)$ holds

(correct evaluation) $\Pi(A, B) = \Pi(A, B)$ is derivable by using first the Π-formation rule and then the reflexivity on types rule.

(parts) The parts are J_1 and J_2 which are assumed to be computable.
Subcase $\Gamma \neq \emptyset$.

(substitution :=) Consider any c.c.s. $a_1 : A_1, \ldots, a_n : A_n$ fitting with $\Gamma \equiv [x_1 : A_1, \ldots, x_n : A_n]$, then

$$A \text{ type } [\Gamma][x_1 := a_1, \ldots, x_n := a_n]$$

is computable with computational complexity lower then α;

$$B(x) \text{ type } [\Gamma, x : A][x_1 := a_1, \ldots, x_n := a_n]$$

is computable, by head substitution lemma 3.8.8, hence the result follows by inductive hypothesis (case 1).

(substitution $\leftarrow$) Consider any c.c.s. $a_1 = a'_1 : A_1, \ldots, a_n = a'_n : A_n$ fitting with the context $\Gamma \equiv [x_1 : A_1, \ldots, x_n : A_n]$. Then

$$A \text{ type } [\Gamma][x_1 \leftarrow a_1 = a'_1, \ldots, x_n \leftarrow a_n = a'_n]$$

is computable with computational complexity lower then α;

$$B(x) \text{ type } [\Gamma, x : A][x_1 \leftarrow a_1 = a'_1, \ldots, x_n \leftarrow a_n = a'_n]$$

is computable, by head substitution lemma, hence the result follows by inductive hypothesis (case 2).

Case 2.

Subcase $\Gamma = \emptyset$.

(associate judgements) The judgements A type and $B(x)$ type $[x : A]$, associate of J_1 and J_2 are computable by assumption, and, by point (3.i) of lemma 3.9.3 also C type and $D(x)$ type $[x : A]$ are computable with the same computational complexity. By point 3.iv of the same lemma, we know that also $D(x)$ type $[x : C]$ is computable and hence the result follows by inductive hypothesis (case 1).

(parts) They are J_1 and J_2 which are assumed to be computable.

Subcase $\Gamma \neq \emptyset$.

(associate judgement) The judgements A type $[\Gamma]$ and $B(x)$ type $[\Gamma, x : A]$ associate respectively of J_1 and J_2 are computable. Hence the result follows by inductive hypothesis (case 1) since the computational complexity of the judgement A type $[\Gamma]$ is lower than that of J_1.

(substitution :=) and (substitution $\leftarrow$) Similar to the previous case 1 by using inductive hypothesis (case 2).

Lemma 3.9.7 (Π-introduction rules) *The* Π*-introduction rules preserve computability. That is*

1. *If* $J_1 \equiv b(x) \in B(x)\ [\Gamma, x : A]$, $J_2 \equiv A$ type $[\Gamma]$ *and* $J_3 \equiv B(x)$ type $[\Gamma, x : A]$ *are computable judgements then*

$$\lambda(b) \in \Pi(A, B)\ [\Gamma]$$

   is computable.

2. *If* $J_1 \equiv b(x) = d(x) \in B(x)\ [\Gamma, x : A]$, $J_2 \equiv A$ type$[\Gamma]$ *and* $J_3 \equiv B(x)$ type $[\Gamma, x : A]$ *are computable judgements then the judgement*

$$\lambda(b) = \lambda(d) \in \Pi(A, B)\ [\Gamma]$$

   is computable.

Proof. By induction on the computational complexity α of J_2.

Case 1.

(associate judgement) The computability of the associate judgements is immediate by the previous lemma 3.9.6 on Π-formation rules.

Subcase $\Gamma = \emptyset$.

(evaluation) $\lambda(b) \Rightarrow \lambda(b)$ holds.

(correct evaluation) The judgement $\lambda(b) = \lambda(b) \in \Pi(A, B)$ is derivable by using first the Π-introduction rule and then the reflexivity on elements rule.

(parts) The part is J_1 which is assumed to be computable.

Subcase $\Gamma \neq \emptyset$.

(substitution :=) Consider any c.c.s. $a_1 : A_1, \ldots, a_n : A_n$ fitting with $\Gamma \equiv [x_1 : A_1, \ldots, x_n : A_n]$, then

$$A \text{ type } [\Gamma][x_1 := a_1, \ldots, x_n := a_n]$$

is computable with computational complexity lower then α;

$$B(x) \text{ type } [\Gamma, x : A][x_1 := a_1, \ldots, x_n := a_n]$$

is computable, by head substitution lemma 3.8.8,

$$b(x) \in B(x)\ [\Gamma, x : A][x_1 := a_1, \ldots, x_n := a_n]$$

is computable, by head substitution lemma 3.8.8, hence the result follows by inductive hypothesis (case 1).

(substitution $\leftarrow$) Consider any c.c.s. $a_1 = a_1' : A_1, \ldots, a_n = a_n' : A_n$ fitting with the context $\Gamma \equiv [x_1 : A_1, \ldots, x_n : A_n]$, then also $a_1 : A_1, \ldots, a_n : A_n$ is a c.c.s. fitting with Γ, and so

$$A \text{ type } [\Gamma][x_1 := a_1, \ldots, x_n := a_n]$$

is computable with computational complexity lower then α;

$$B(x) \text{ type } [\Gamma, x : A][x_1 := a_1, \ldots, x_n := a_n]$$

is computable, by head substitution lemma 3.8.8, and

$$b(x) \in B(x) \ [\Gamma, x : A][x_1 \leftarrow a_1 = a_1', \ldots, x_n \leftarrow a_n = a_n']$$

is computable, by head substitution lemma 3.8.8, hence the result follows by inductive hypothesis (case 2).

Case 2.

Subcase $\Gamma = \emptyset$.

(associate judgements) The judgement $b(x) \in B(x) \ [x : A]$, which is the associate of J_1, is computable by definition and thus, by point 4.i of lemma 3.9.3, also $d(x) \in B(x) \ [x : A]$ is computable. Hence the result follows by case 1.

(parts) The part judgement is J_1 which is assumed to be computable.

Subcase $\Gamma \neq \emptyset$.

(associate judgements) The judgement $b(x) \in B(x) \ [x : A]$, associate of J_1 is computable by definition, hence the result follows by case 1.

(substitutions :=) and (substitution $\leftarrow$) Similar to the previous case 1 by using inductive hypothesis (case 2).

Lemma 3.9.8 (Π-elimination rules) *The* Π*-elimination rules preserve computability. That is*

1. *If the judgements* $J_1 \equiv c \in \Pi(A, B) \ [\Gamma]$, $J_2 \equiv d(y) \in C(\lambda(y)) \ [\Gamma, y : (x : A)B(x)]$ *and* $J_3 \equiv C(t)$ type $[\Gamma, t : \Pi(A, B)]$ *are computable then the judgement*

$$\mathsf{F}(c, d) \in C(c) \ [\Gamma]$$

is computable.

2. *If the judgements* $J_1 \equiv c = c' \in \Pi(A, B) \ [\Gamma]$, $J_2 \equiv d(y) = d'(y) \in C(\lambda(y)) \ [\Gamma, y : (x : A)B(x)]$ *and* $J_3 \equiv C(t)$ type $[\Gamma, t : \Pi(A, B)]$ *are computable then the judgement*

$$\mathsf{F}(c, d) = \mathsf{F}(c', d') \in C(c) \ [\Gamma]$$

is computable.

Proof. By induction on the computational complexity α of J_1.

Case 1.

(associate judgements) The computability of the judgement $C(c)$ type $[\Gamma]$, associate of the judgement $\mathsf{F}(c, d) \in C(c) \ [\Gamma]$, follows by substitution lemma 3.8.9.

Subcase $\Gamma = \emptyset$.

(evaluation) J_1 is computable and $\Pi(A, B) \Rrightarrow \Pi(A, B)$, then $c \Rightarrow \lambda(b)$ and the judgement $b : (x : A)B(x)$ is computable; thus it is a c.c.s. fitting with $y : (x : A)B(x)$; therefore $J_2[y := b]$, which is $d(b) \in C(\lambda(b))$, is a computable judgement. Hence $d(b) \Rightarrow g$ and the result follows by using the computation rule.

(correct evaluation) Since J_1 is computable, we know that there exists a derivation of the judgement $\Pi(A, B)$ type; hence $x : \Pi(A, B)$ is a correct assumption, and $c = \lambda(b) \in \Pi(A, B)$ is derivable. Let Π_1 be the following derivation

$$\dfrac{\dfrac{c = \lambda(b) \in \Pi(A, B) \quad \dfrac{x \in \Pi(A, B) \ [x : \Pi(A, B)] \quad J_2 \quad J_3}{\mathsf{F}(x, d) \in C(x) \ [x : \Pi(A, B)]}}{\mathsf{F}(c, d) = \mathsf{F}(\lambda(b), d) \in C(c)} \quad \dfrac{c = \lambda(b) \in \Pi(A, B) \quad J_3}{C(c) = C(\lambda(b))}}{\mathsf{F}(c, d) = \mathsf{F}(\lambda(b), d) \in C(\lambda(b))}$$

Since J_1 is computable, so is $b(x) \in B(x)\ [x : A]$ and then $J_2[y := b]$ is computable. Thus the judgements $d(b) = g \in C(\lambda(b))$ and $b(x) \in B(x)\ [x : A]$ are derivable. Let Π_2 be the following derivation

$$\dfrac{\dfrac{\Pi_1}{\mathsf{F}(c,d) = \mathsf{F}(\lambda(b),d) \in C(\lambda(b))} \quad \dfrac{\dfrac{b(x) \in B(x)\ [x : A] \quad \Pi(A,B)\ \mathsf{type} \quad J_2 \quad J_3}{\mathsf{F}(\lambda(b),d) = d(b) \in C(\lambda(b))} \qquad d(b) = g \in C(\lambda(b))}{\mathsf{F}(\lambda(b),d) = g \in C(\lambda(b))}}{\mathsf{F}(c,d) = g \in C(\lambda(b))}$$

Hence

$$\dfrac{\dfrac{\Pi_2}{\mathsf{F}(c,d) = g \in C(\lambda(b))} \quad \dfrac{C(c) = C(\lambda(b))}{C(\lambda(b)) = C(c)}}{\mathsf{F}(c,d) = g \in C(c)}$$

(parts) Since J_1 is computable we know that $c \Rightarrow \lambda(b)$ and, by point 3 of fact 3.8.3, that the judgement $\lambda(b) \in \Pi(A,B)$ is computable. Hence, by point i of lemma 3.8.7, we can deduce that the judgement $c = \lambda(b) \in \Pi(A,B)$ is computable. Therefore, since J_3 is computable, we obtain that the judgement $C(\lambda(b)) = C(c)$ is a computable. Then, since $d(b) \in C(\lambda(b))$ is a computable judgement, so is $d(b) \in C(c)$, by point 3.iii of lemma 3.9.3. Hence, since $d(b) \Rightarrow g$, the parts of g, which is also the value of $\mathsf{F}(c,d)$, are computable element(s) in the value of $C(c)$.
Subcase $\Gamma \neq \emptyset$

(substitution :=) Immediate by inductive hypothesis (case 1).

(substitution $\leftarrow$) Immediate by inductive hypothesis (case 2).

Case 2.

(associate judgements) The computability of $\mathsf{F}(c,d) \in C(c)\ [\Gamma]$, which is the judgement associate of $\mathsf{F}(c,d) = \mathsf{F}(c',d') \in C(c)\ [\Gamma]$, follows by case 1. If Γ is empty, also the computability of $\mathsf{F}(c',d') \in C(c')$ follows by case 1. since from the fact that J_1 and J_2 are computable, by point 4.i of lemma 3.9.3, we obtain that $c' \in \Pi(A,B)$ and $d'(y)' \in C(\lambda(y))\ [y : (x : A)B(x)]$ are computable judgements. Then, since the judgement $C(c) = C(c')$ is computable, by point 3.iii.a of lemma 3.9.3, $\mathsf{F}(c',d') \in C(c')$ is computable.
Subcase $\Gamma = \emptyset$.

(parts) The judgement J_1 is computable, then $c \Rightarrow \lambda(b)$, $c' \Rightarrow \lambda(b')$ and the judgement $b(x) = b'(x) \in B(x)\ [x : A]$ is computable. Moreover $b = b' : (x : A)B(x)$ is a c.c.s. for $y : (x : A)B(x)$ in J_2, and then $J_2[y \leftarrow b = b']$, that is the judgement $d(b) = d'(b') \in C(\lambda(b))$ is computable. Then, by point 3.iii.b of lemma 3.9.3, $d(b) = d'(b') \in C(c)$ is a computable judgement, since, as in the previous point, we can prove that $C(\lambda(b)) = C(c)$ is a computable judgement. So, if $d(b) \Rightarrow g_d$ and $d'(b') \Rightarrow g_{d'}$, the parts of g_d and $g_{d'}$ are computable equal elements in the value of $C(c)$.
Subcase $\Gamma \neq \emptyset$.

(substitution :=) and (substitution $\leftarrow$) Immediately follow by inductive hypothesis (case 2.).

Lemma 3.9.9 (Π-equality rule) *The Π-equality rule preserves computability, i.e., if the judgements $J_1 \equiv b(x) \in B(x)\ [\Gamma, x : A]$, $J_2 \equiv \Pi(A,B)\ \mathsf{type}\ [\Gamma]$, $J_3 \equiv d(y) \in C(\lambda(y))\ [\Gamma, y : (x : A)B(x)]$ and $J_4 \equiv C(t)\ \mathsf{type}\ [\Gamma, t : \Pi(A,B)]$ are computable then the judgement*

$$\mathsf{F}(\lambda(b),d) = d(b) \in C(\lambda(b))\ [\Gamma]$$

is computable.

Proof. The proof is by induction on the computational complexity α of J_2.

(associated judgements) J_1 and J_2 are computable judgements by assumption. Thus, lemma 3.9.7 on Π-introduction rules yields that $\lambda(b) \in \Pi(A,B)\ [\Gamma]$ is a computable judgement, and hence $\mathsf{F}(\lambda(b),d) \in C(\lambda(b))\ [\Gamma]$ is computable by the previous lemma on Π-elimination rules. Moreover, if Γ is empty, since J_1 and J_3 are computable, we obtain that $J_3[y := b]$, i.e. the second associate judgement $d(b) \in C(\lambda(b))$, is computable.
Subcase $\Gamma = \emptyset$.

Since $\mathsf{F}(\lambda(b), d)$ and $d(b)$ evaluate into the same canonical element, the computability of the judgement $\mathsf{F}(\lambda(b), d) = d(b) \in C(\lambda(b))$ follows from the computability of the associated judgements. Subcase $\Gamma \neq \emptyset$.

(substitution :=) It immediately follows by inductive hypothesis.

(substitution $\leftarrow$) Consider any c.c.s. $a_1 = a_1' : A_1, \ldots, a_n = a_n' : A_n$ fitting with the context $\Gamma \equiv [x_1 : A_1, \ldots, x_n : A_n]$, then also $a_1 : A_1, \ldots, a_n : A_n$ is a c.c.s. fitting with Γ, and, by inductive hypothesis we obtain that

$$\mathsf{F}(\lambda(b), d) = d(b) \in C(\lambda(b)) \ [\Gamma][x_1 := a_1, \ldots, x_n := a_n]$$

is computable.

Moreover, since $J_3[y := b]$ is computable, also $J_3[y := b][x_1 \leftarrow a_1 = a_1', \ldots, x_n \leftarrow a_n = a_n']$, which is

$$d(b) \in C(\lambda(b)) \ [\Gamma][x_1 \leftarrow a_1 = a_1', \ldots, x_n \leftarrow a_n = a_n']$$

is computable and then the result follows by transitivity.

For all the other cases, with a few exceptions, the proof goes on analogously to the Π case. In the following we will stress the essential points.

- We proceed always by induction on the computational complexity of the first premise such that none of its assumptions is discharged.
- For each type we must consider the rules in the following association and ordering:
  - the two formation rules
  - the two introduction rules
  - the two elimination rules (with the exception of the cases of the Eq and U types)
  - the equality rule.

  The ordering is important since, in some cases, to carry on the proof we need to apply a rule which precedes the considered one in the given ordering and therefore we must have already proved that such a rule preserves computability. For instance, when the first introduction rule is considered, the computability of the associate judgement follows by applying the formation rule to some suitable judgements which are among the premises.

  The association is important since, when the first of the two associated rules is considered, to prove the computability of the substituted judgements (substitution $\leftarrow$) we apply the second rule while, when the second rule is considered, the computability of the associate judgements follows by applying the first rule.

(associate judgements)

- As regard to the first introduction rule, the computability of the associate judgements follows, as already noted, by applying the formation rule to suitable judgements which are among the premises.
- As regard to the first elimination rule, the computability of the associate judgements follows by applying a suitable substitution to one of the premise. U-elimination rules are different and they had been treated in Lemma 3.9.2.
- As regard to the second formation, introduction or elimination rule, the computability of the associate judgements follows, by inductive hypothesis, by applying the first rule to the associate of the premises or to their variants whose computability is assured by definition or by lemma 3.9.3 and also lemma 3.9.2 when U-introductions are considered. These lemmas are needed in order to prove the computability of the second associate judgement or to allow switching the assumptions from one type to a computationally equal one. For instance, in the Π-case from the computability of the judgements $B(x) = D(x) \ [x : A]$ and $A = C$, we deduced, by points 3.i and 3.iv of lemma 3.9.3, the computability of $D(x) \ \mathsf{type} \ [x : C]$

which is a variant of the computationally equal judgement $B(x)$ type $[x : A]$. Only for the elimination rules, in the case $\Gamma \neq \emptyset$, the application of the first rule does not immediately produce the wanted associate: a changing of type is required, and allowed by lemma 3.9.3, since $C(c) = C(c')$ is a computable judgement. Clearly Eq-elimination is an exception (there is only one elimination rule). In this case, when a substitution $e = f :$ is considered in order to prove the computability of a hypothetical judgement $a = b \in A$ $[\Gamma]$ derived from the computable premises $c \in \mathsf{Eq}(A, a, b)$ $[\Gamma]$, A type $[\Gamma]$, $a \in A$ $[\Gamma]$, $b \in A$ $[\Gamma]$, the computability of the saturated judgement can be proved as follows. The substitution $e :$ is first applied to the premises in order to obtain, by inductive hypothesis, that the judgement $a = b \in A$ $[\Gamma][.. := e]$ is computable; then the substitution $e = f :$ is applied to the judgement $b \in A$ $[\Gamma]$; the result follows by transitivity (point 1.1 of lemma 3.9.3).

- As regard to the equality rule, the computability of the first associate is obtained by using an instance of the introduction rule and an instance of the elimination rule of the considered type. In the case $\Gamma = \emptyset$, the computability of the other associate judgement is obtained by a suitable use of the substitution rules, that is easy, even if not immediate, also in the case of the inductive types N and W. The only exception is the type U where suitable formation rules, that preserves computability (see Lemma 3.9.2), must be applied to the judgements of kind type that one obtain by using the first U-elimination rule.

(evaluation), (correct evaluation), (parts)

- When formation or introduction rules are considered, the points (evaluation), (correct evaluation), (parts), are always immediate.

- As regards elimination rule, the points (evaluation), (correct evaluation), (parts), in the case $\Gamma = \emptyset$, must be a little more detailed.

  Case 1: first elimination rule.

  Let $\mathsf{non-can-el} \in C(c)$ be the conclusion of the considered rule (in the Π-case we have $\mathsf{F}(c, d) \in C(c)$). First of all, note that there is always a premise of the form $c \in \mathsf{Tp}$ where the outermost constant of the expression Tp characterizes the type to which the elimination refers (in the Π-case we have $c \in \Pi(A, B)$), a hypothetical type-judgement depending on Tp (in the Π-case we have $C(t)$ type $[t : \Pi(A, B)]$) and one or more other minor premises (in the Π-case we have $d(y) \in C(\lambda(y))$ $[y : (x : A)B(x)]$). Then the proof gets on in the following way.

  - (evaluation) The canonical value g_c of c ($\lambda(b)$ in the Π-case), which exists since the major premise $c \in \mathsf{Tp}$ is computable, allows one to choose which minor premises to analyze (in the Π-case there is only one minor premise, that is $d(y) \in C(\lambda(y))$ $[y : (x : A)B(x)]$). When this is a hypothetical judgement it must be saturated and the part judgements of the major premise gives us some of the substitutions needed to saturate it (in the Π-case we obtained $d(b) \in C(\lambda(b))$). This saturated judgement, $\mathsf{sat-el} \in C(g_c)$ is computable and its evaluation is exactly what we are looking for. Usually the parts of the major premise together with the other premises provides all the needed substitutions; exceptions are the cases U, which had been considered in lemma 3.9.2, N and W where an induction on the complexity of the major premise is necessary to build the suitable substitution. Let us develop these two cases in detail.

    * N-elimination. The premises are $c \in \mathsf{N}$, $d \in C(0)$, $e(x, y) \in C(\mathsf{s}(x))$ $[x : \mathsf{N}, y : C(x)]$ and $C(t)$ type $[t : \mathsf{N}]$ Then, $c \in \mathsf{N}$ is computable; thus either $c \Rightarrow 0$ or $c \Rightarrow \mathsf{s}(a)$.
      If $c \Rightarrow 0$ then we choose $d \in C(0)$ among the minor premises and the value of d, which exists since $d \in C(0)$ is computable, is just the value of $\mathsf{N}_{rec}(c, d, e)$. Otherwise, if $c \Rightarrow \mathsf{s}(a)$, we choose $e(x, y) \in C(\mathsf{s}(x))$ $[x : \mathsf{N}, y : C(x)]$. Now, $a \in \mathsf{N}$ is computable; then $a : \mathsf{N}$ is a c.c.s. fitting with $x : \mathsf{N}$. Moreover, $\mathsf{comp}(a \in \mathsf{N}) < \mathsf{comp}(c \in \mathsf{N})$; thus, by inductive hypothesis, $\mathsf{N}_{rec}(a, d, e) \in C(a)$ is computable and hence $a : \mathsf{N}$ together with $\mathsf{N}_{rec}(a, d, e) : C(a)$ is a c.c.s. fitting with $x : \mathsf{N}$, $y : C(x)$. Hence $e(a, \mathsf{N}_{rec}(a, d, e)) \in C(\mathsf{s}(a))$ is computable and the value of $e(a, \mathsf{N}_{rec}(a, d, e))$ is just the value of $\mathsf{N}_{rec}(c, d, e)$.

* W-elimination. Note that the judgment $c \in \mathsf{W}(A,B)$ is computable by assumption; then $c \Rightarrow \mathsf{sup}(a,b)$ and $a \in A$ and $b(x) \in \mathsf{W}(A,B)$ $[x : B(a)]$, that is $b : (x : B(a))\ \mathsf{W}(A,B)$, are computable judgements. Hence we obtaind that $a : A$, $b : (x : B(a))\ \mathsf{W}(A,B)$ is a c.c.s. fitting with $z : A$, $y : (x : B(z))\ \mathsf{W}(A,B)$. Then, by applying a W-elimination rule on $b(x) \in \mathsf{W}(A,B)[x : B(a)]$ instead of $c \in \mathsf{W}(A,B)$ we obtain by inductive hypothesis that $\mathsf{T}_{rec}(b(x),d) \in C(b(x))$ $[x : B(a)]$ is a computable judgement, because the computational complexity of the judgement $b(x) \in \mathsf{W}(A,B)$ $[x : B(a)]$ is lower than the computational complexity of the judgement $c \in \mathsf{W}(A,B)$. Now, note that the judgement $\mathsf{T}_{rec}(b(x),d) \in C(b(x))$ $[x : B(a)]$ is equivalent by definition to $(x)\ \mathsf{T}_{rec}(b(x),d) : (x : B(a))\ C(b(x))$ and hence it is a c.c.s. fitting with $t : (x : B(a))\ C(b(x))$. Then, by substituting, we obtain that $d(a,b,(x)\ \mathsf{T}_{rec}(b(x),d)) \in C(\mathsf{sup}(a,b))$ is computable and the value of $d(a,b,(x)\ \mathsf{T}_{rec}(b(x),d))$ is exactly the value of $\mathsf{T}_{rec}(c,d)$ we are looking for.

– (correct evaluation) For each canonical value of the major premise a derivation can be constructed analogously to what we did in the Π-case. It is sufficient to substitute any application of Π-elimination and Π-equality rules by the corresponding one for the considered type.

– (parts) The computability of the major premise $c \in \mathsf{Tp}$ guarantees, by lemma 3.8.3 and point i of lemma 3.8.7, the computability of the judgement $c = g_c \in \mathsf{Tp}$ (in the Π-case we had $c = \lambda(b) \in \Pi(A,B)$). This fact yields that the type equality judgement $C(c) = C(g_c)$ is computable (in the Π-case we proved that $C(c) = C(\lambda(b))$ is computable). Now, if we consider the computable judgement built up to prove the previous evaluation point, $\mathsf{sat} - \mathsf{el} \in C(g_c)$ (in the Π-case we had $d(b) \in C(\lambda(b))$), by point 3.iii of lemma 3.9.3, we obtain that $\mathsf{sat} - \mathsf{el} \in C(c)$ is computable (in the Π-case we had $d(b) \in C(c)$). Hence if $\mathsf{sat} - \mathsf{el} \Rightarrow \mathsf{can} - \mathsf{el}$ and $C(c) \Rightarrow \mathsf{can}_C$, then also $\mathsf{non} - \mathsf{can} - \mathsf{el} \Rightarrow \mathsf{can} - \mathsf{el}$ and the parts of $\mathsf{can} - \mathsf{el}$ are computable elements in can_C.

Case 2: second elimination rule.

Let $\mathsf{non} - \mathsf{can} - \mathsf{el}_1 = \mathsf{non} - \mathsf{can} - \mathsf{el}_2 \in C(c)$ be the conclusion of the rule.

(parts) First of all, note that the computability of the first associate of the major premise, $c = c' \in \mathsf{Tp}$, guarantees the computability of the judgement $C(c) = C(g_c)$. Then, analogously to the case 1 of the first elimination rule, we can choose the suitable minor premise and saturate it by using $\leftarrow$ instead of $:=$. By the computability of the resulting judgement $\mathsf{sat} - \mathsf{el}_1 = \mathsf{sat} - \mathsf{el}_2 \in C(g_c)$ together with that of $C(c) = C(g_c)$, we will obtain the computability of $\mathsf{sat} - \mathsf{el}_1 = \mathsf{sat} - \mathsf{el}_2 \in C(c)$. From this the result is immediate.

- For the equality rule, the point (parts), follow easily since, by point i of lemma 3.8.7 (or point ii when the type U is considered), the computability of the associate judgements together with the definition of the evaluation process $\Rightarrow$, guarantees the computability of the judgement in the conclusion.

(substitution $:=$)

- The point (substitution $:=$) always follows, by induction, by first applying the same substitution to the premises and next applying again the same rule to the resulting judgements. Note that when a rule which discharges assumptions is considered, we must apply a head substitution which preserves computability.

(substitution $\leftarrow$)

- For the first formation, introduction or elimination rule, the point (substitution $\leftarrow$) follows, by inductive hypothesis, by applying the second rule in the association to judgements obtained by properly substituting the given premises. In some cases, when a rule which discharges assumptions is considered, the computability of the suitably substituted premises is stated by the head substitution lemma 3.8.8.

- For the second formation, introduction or elimination rule, the proof of the point (substitution $\leftarrow$) follows by applying the same rule to judgements obtained by wisely applying the same substitution $e = f :$ or its associate $e :$ to the given premises.
- For the equality rule, when the substitution $e = f :$ is considered in proving the point (substitution $\leftarrow$), we will proceed as follows. On one side we apply the same rule to judgements obtained by applying the associate substitution $e :$ to the premises. On the other side, we apply the substitution $e = f :$ to a judgement built up by applying a suitable head substitution to the minor premise analogously to what done for the (evaluation) point. The result then follows by transitivity (consider again the Π-case as a typical example). For the U-case we must build up a first judgement by applying the same rule to judgements obtained by applying the substitution $e :$ to the given premises, and a second one by applying a formation rule to the result of applying the U-elimination to the premises. The result then follows by transitivity. Note that all the rules used in the construction preserve computability.

3.10 The computability theorem

Now we can state our main theorem: it shows that any derivable judgement is computable and hence that all the properties we ask for a judgement to be computable hold for any derivable judgement.

Theorem 3.10.1 (Computability theorem) *Let J be any derivable judgement. Then J is computable.*

From a proof-theoretical point of view the main meta-theoretical result on a deductive system in natural deduction style as ours, is a normal form theorem, i.e. a theorem that states that any proof can be transformed in a new one with the same conclusion but enjoying stronger structure properties. These properties generally allow in turn to deduce important properties on the considered deduction system such as its consistency. Our computability theorem does not regard derivations but still is strongly related to normal form theorems as the following definitions will clarify.

Definition 3.10.2 (Canonical proof) *A proof Π of a judgement J is* canonical *if*

1. *$J \equiv A$ type or $J \equiv A = B$ and the last inference step in Π is a formation rule.*
2. *$J \equiv a \in A$ or $J \equiv a = b \in A$ and the last inference step in Π is an introduction rule.*

A canonical proof might be also called "normal at the end". Clearly not every closed judgement can be derived by a canonical proof. This holds only for the judgements which, according to the following definition, are in canonical form.

Definition 3.10.3 (Canonical form) *Let J be a closed judgement. Then*

- *if $J \equiv A$ type and $A \Rrightarrow G_A$ then the* canonical form *of J is G_A type;*
- *if $J \equiv A = B$ and $A \Rrightarrow G_A$ and $B \Rrightarrow G_B$ then the* canonical form *of J is $G_A = G_B$;*
- *if $J \equiv a \in A$ and $a \Rightarrow g_a$ and $A \Rrightarrow G_A$ then the* canonical form *of J is $g_a \in G_A$;*
- *if $J \equiv a = b \in A$ and $a \Rightarrow g_a$ and $b \Rightarrow g_b$ and $A \Rrightarrow G_A$ then the* canonical form *of J is $g_a = g_b \in G_A$.*

Corollary 3.10.4 (Canonical-form theorem) *Let J be a derivable closed judgement then there exists a canonical proof of the canonical form of J.*

Proof. Since J is derivable then it is computable and hence there exist a derivation of its parts judgements since they also are computable. By putting them together with a formation or an introduction rule we obtain a canonical proof of the canonical form of J.

It is easy to see that if J is a derivable closed judgement then its computability implies that its canonical form is a judgement equivalent to J, in fact:

- if $J \equiv A$ type and $A \Rrightarrow G_A$ then the canonical form of J is G_A type and the computability of J assures that $A = G_A$ is derivable.
- if $J \equiv A = B$ and $A \Rrightarrow G_A$ and $B \Rrightarrow G_B$ then the canonical form of J is $G_A = G_B$ and the computability of J assures that $A = G_A$ and $B = G_B$ are derivable judgements.
- if $J \equiv a \in A$ and $a \Rightarrow g_a$ and $A \Rrightarrow G_A$ then the canonical form of J is $g_a \in G_A$ and the computability of J yields that $A = G_A$ and $a = g_a \in A$ are derivable judgements.
- if $J \equiv a = b \in A$ and $a \Rightarrow g_a$, $b \Rightarrow g_b$ and $A \Rrightarrow G_A$ then the canonical form of J is $g_a = g_b \in G_A$ and the computability of J assures that $A = G_A$, $a = g_a \in A$ and $b = g_b \in A$ are derivable judgements.

Then the previous canonical form theorem is, in our system, the counterpart of a standard normal form theorem since it guarantees that if J is a closed derivable judgement then we can construct a canonical proof for a judgement equivalent to J. Moreover it allows us to deduce most of the results usually obtained by a normal form theorem such as, for instance, consistency.

Corollary 3.10.5 (Consistency of HITT) *The Higher order Intuitionistic Theory of Type is consistent.*

Proof. Since the judgement $c \in \mathsf{N}_0$ is not computable and then it cannot be derivable.

Note that this result establishes also the consistency of the original ITT. As we could expect, the minimal properties, which are usually asked for a logical system to be considered constructive, immediately follow just by reading the definition of computable judgement.

Corollary 3.10.6 (Disjunction property) *If the judgement $c \in +(A, B)$ is derivable then either there exists an element a such that $a \in A$ is derivable or there exists an element b such that $b \in B$ is derivable.*

Proof. If $c \in +(A, B)$ is derivable then it is computable and hence either $c \Rightarrow \mathsf{inl}(a)$ and $a \in A$ is derivable or $c \Rightarrow \mathsf{inr}(b)$ and $b \in B$ is derivable.

Corollary 3.10.7 (Existential property) *If the judgement $c \in \Sigma(A, B)$ is derivable then there exists an element b such that the judgement $b \in B(a)$ is derivable for some $a \in A$.*

Proof. If $c \in \Sigma(A, B)$ is derivable then it is computable and hence $c \Rightarrow \langle a, b\rangle$ and $a \in A$ and $b \in B(a)$ are derivable judgements.

Other consequences of the computability theorem can be stated when the Intuitionistic Theory of Type is viewed as a formal system to derive programs, that is when a type is interpreted as the specification of a problem and an element in this type as the program which meets this specification. In this environment an expression denoting an element in a type is thought as a program written in a functional language whose operational semantics is given by the computation rules and hence to execute a program corresponds to evaluating it. The computability theorem shows that whenever we prove that a program a is partially correct with respect to its specification A, i.e. we derive the judgement $a \in A$, then we know also that it is totally correct, that is its evaluation terminates.

Corollary 3.10.8 (Evaluation theorem) *Any provable expression can be evaluated, that is*

1. *If A type is a provable judgement then A has a canonical value.*
2. *If $a \in A$ is a provable judgement, then a has a canonical value.*

Thus any program whose evaluation does not terminate, such as the famous Church's non-terminating function, cannot be typed in HITT.

Related works. We should like to thank the referee for his comments and suggestions and for pointing out to us related works, in particular [All86] and [All87] where similar results are proved by a realizability-like semantics.

Chapter 4

Properties of Type Theory

4.1 Summary

In this chapter some meta-mathematical properties of Martin-Löf's type theory will be presented.

In the first part we will show that decidability of a property over elements of a set A can be reduced to the problem of finding a suitable function from A into the set Boole ($\equiv \mathsf{N}_2$).

Then an intuitionistic version of Cantor's theorem will be shown which proves that there is no bijection between the set N of the natural numbers and the set $\mathsf{N} \to \mathsf{N}$ of the functions from natural numbers into natural numbers.

Finally, the "forget-restore" principle will be illustrated by analyzing a simple case in Martin-Löf's type theory. Indeed, type theory offers a way of "forgetting" information, that is, supposing A set, the form of judgment A true. The meaning of A true is that there exists an element a such that $a \in A$ but it does not matter which particular element a is (see also the notion of *proof irrelevance* in [Bru80]). Thus to pass from the judgment $a \in A$ to the judgment A true is a clear example of the forgetting process. In this section we will show that it is a constructive way to forget since, provided that there is a proof of the judgment A true, an element a such that $a \in A$ can be re-constructed.

4.2 Decidability is functionally decidable

Introduction and basic lemmas

In this section we show that the usual intuitionistic characterization of the decidability of the propositional function $B(x)$ prop $[x : A]$, that is to require that the predicate $(\forall x \in A)\ B(x) \vee \neg B(x)$ is provable, is equivalent, when working within the framework of Martin-Löf's Intuitionistic Type Theory, to require that there exists a *decision function* $\phi : A \to \mathsf{Boole}$ such that

$$(\forall x \in A)\ (\phi(x) =_{\mathsf{Boole}} \mathsf{true}) \leftrightarrow B(x)$$

Since we will also show that the proposition $x =_{\mathsf{Boole}} \mathsf{true}$ $[x : \mathsf{Boole}]$ is decidable, we can alternatively say that the main result of this section is a proof that the decidability of the predicate $B(x)$ prop $[x : A]$ can be effectively reduced by a function $\phi \in A \to \mathsf{Boole}$ to the decidability of the predicate $\phi(x) =_{\mathsf{Boole}} \mathsf{true}$ $[x : A]$. All the proofs are carried out within the Intuitionistic Type Theory and hence the decision function ϕ, together with a proof of its correctness, is effectively constructed as a function of the proof of $(\forall x \in A)\ B(x) \vee \neg B(x)$.

This result may not be completely new (for instance in a personal communication Martin-Löf said that he already knew it) but since, to my knowledge, there is no published material on this topic this note may be useful to a wider audience. In fact, apart from its intrinsic relevance, this result is also a good exercise in ITT since in order to be able to obtain its proof one has to use some of the most interesting properties of ITT. In this paragraph we will recall these properties, and their proofs, to the reader who is not familiar with ITT, but to avoid to bore the reader who

is familiar with ITT we will not recall all the basic definitions which can be found in [Mar84] or [NPS90].

Since in most of the results the type Boole plays a central role, let us begin by recalling some of its properties. First of all recall that the canonical elements of the type Boole are true and false and that, supposing $C(x)$ prop $[x : \mathsf{Boole}]$, the Boole-elimination rule allows to obtain

$$\text{if } c \text{ then } d \text{ else } e \in C(c)$$

provided $c \in \mathsf{Boole}$, $d \in C(\mathsf{true})$ and $e \in C(\mathsf{false})$.

Lemma 4.2.1 *Let* $P(x)$ prop $[x : \mathsf{Boole}]$ *and* $c \in \mathsf{Boole}$*; then*

$$P(c) \to P(\mathsf{true}) \vee P(\mathsf{false})$$

Proof. The proof is just an application of the Boole-elimination rule. In fact if $c \in \mathsf{Boole}$ then

$$\text{if } c \text{ then } \lambda x.\mathsf{inl}(x) \text{ else } \lambda y.\mathsf{inr}(y) \in P(c) \to P(\mathsf{true}) \vee P(\mathsf{false})$$

since $\lambda x.\mathsf{inl}(x) \in P(\mathsf{true}) \to P(\mathsf{true}) \vee P(\mathsf{false})$ and $\lambda y.\mathsf{inr}(y) \in P(\mathsf{false}) \to P(\mathsf{true}) \vee P(\mathsf{false})$.

Let us recall that, supposing $c, d \in \mathsf{Boole}$, by $c =_{\mathsf{Boole}} d$ we mean the intensional equality proposition for elements of Boole, denoted in the appendix B by the type Id. Its main properties are that

- if $c = d \in \mathsf{Boole}$ then $c =_{\mathsf{Boole}} d$ is true,
- if $c =_{\mathsf{Boole}} d$ is true and $A(x)$ prop $[x : \mathsf{Boole}]$ is a proposition on elements of Boole such that $a \in A(c)$ then $\mathsf{move}(c, a) \in A(d)$,
- if $c =_{\mathsf{Boole}} d$ is true, $a(x) \in A$ $[x : \mathsf{Boole}]$ and $y =_A z$ prop $[y, z : A]$ is the intensional equality proposition for elements of the type A then $a(c) =_A a(d)$ is true.

Hence the following corollary is immediate.

Corollary 4.2.2 *Let* $c \in \mathsf{Boole}$*; then* $(c =_{\mathsf{Boole}} \mathsf{true}) \vee (c =_{\mathsf{Boole}} \mathsf{false})$.

Proof. Suppose $P(x) \equiv c =_{\mathsf{Boole}} x$ $[x : \mathsf{Boole}]$, then the previous lemma shows that

$$(c =_{\mathsf{Boole}} c) \to (c =_{\mathsf{Boole}} \mathsf{true}) \vee (c =_{\mathsf{Boole}} \mathsf{false})$$

Now the statement is obvious since $c =_{\mathsf{Boole}} c$ is straightforward.

We have then proved that in Boole there are at most two elements; by means of the universe of the small types U, whose elements are (the codes of) the basic types, we can show that in Boole there are exactly two elements. To obtain this result it is convenient to use the equality proposition $A =_{\mathsf{U}} B$ prop $[A, B : \mathsf{U}]$ for elements of the type U. Besides the properties analogous to those above for the equality proposition for the elements of the type Boole, in this case one can also prove that if $A =_{\mathsf{U}} B$ is true and $a \in A$ then $\mathsf{shift}(a) \in B$.

Lemma 4.2.3 $\neg(\mathsf{true} =_{\mathsf{Boole}} \mathsf{false})$

Proof. Assume that $y : \mathsf{Boole}$. Then a Boole-elimination can be used to shows that

$$\text{if } y \text{ then } \top \text{ else } \bot \in \mathsf{U}$$

where $\top$ is the one-element type and $\bot$ is the empty type.

Let us now assume that $x : \mathsf{true} =_{\mathsf{Boole}} \mathsf{false}$, i.e. let us assume that $\mathsf{true} =_{\mathsf{Boole}} \mathsf{false}$ is true. Then we obtain that

$$\text{if } \mathsf{true} \text{ then } \top \text{ else } \bot =_{\mathsf{U}} \text{if } \mathsf{false} \text{ then } \top \text{ else } \bot$$

and hence $\top =_{\mathsf{U}} \bot$ since the equality proposition is transitive and $\top =_{\mathsf{U}} \text{if } \mathsf{true} \text{ then } \top \text{ else } \bot$ and if false then $\top$ else $\bot =_{\mathsf{U}} \bot$ hold; hence, supposing $*$ is the only element of the type $\top$,

$\mathsf{shift}(*) \in \bot$, that is we have found an element in the empty type; so, by discharging the assumption $x : \mathsf{true} =_{\mathsf{Boole}} \mathsf{false}$, we finally obtain $\lambda x.\ \mathsf{shift}(*) \in \neg(\mathsf{true} =_{\mathsf{Boole}} \mathsf{false})$.

We showed a full detailed proof of this lemma to stress the fact that it is completely carried out within ITT with the universe U of the small types.

By using lemma 4.2.3 and a little of intuitionistic logic one can prove the following not very surprising result.

Lemma 4.2.4 $(c =_{\mathsf{Boole}} \mathsf{false})$ *if and only if* $\neg(c =_{\mathsf{Boole}} \mathsf{true})$.

Even if the previous lemma is straightforward when we combine it with 4.2.2 we obtain an interesting result: the predicate $x =_{\mathsf{Boole}} \mathsf{true}\ [x : \mathsf{Boole}]$ is decidable.

Corollary 4.2.5 *For all* $c \in \mathsf{Boole}$, $(c =_{\mathsf{Boole}} \mathsf{true}) \vee \neg(c =_{\mathsf{Boole}} \mathsf{true})$.

There is another property that we need to recall because of its relevance in the following: thanks to the constructive meaning of the logical connectives a sort of *Axiom of Choice* holds in ITT (here we show a statement which is not the strongest one that can be proved but it is sufficient for us).

Lemma 4.2.6 *Let* A, B *be two types and* $C(x, y)\ \mathsf{prop}\ [x : A, y : B]$*; then*

$$((\forall x \in A)(\exists y \in B)\ C(x, y)) \to ((\exists f \in A \to B)(\forall x \in A)\ C(x, f(x)))$$

Proof. A complete proof can be found in [Mar84] where a choice function $f \in A \to B$ is constructed together with a formal proof that for any $x \in A$, $C(x, f(x))$ holds; anyhow the basic intuition to obtain the proof is rather simple: suppose h is (the code for) a proof of $(\forall x \in A)(\exists y \in B)\ C(x, y)$ then, for any $x \in A$, (the value of) $h(x)$ is a couple whose first element $\mathsf{fst}(h(x))$ belongs to B while the second element is a proof of $C(x, \mathsf{fst}(h(x)))$; the choice function is then $\lambda x.\mathsf{fst}(h(x)) \in A \to B$.

Also in this case we want to observe that the proof explicitly shows how to construct a choice function in $A \to B$ provided that we have a proof of $(\forall x \in A)(\exists y \in B)\ C(x, y)$.

4.2.1 The main result

This paragraph is completely devoted to the proof of the following theorem.

Theorem 4.2.7 *Let* $B(x)\ \mathsf{prop}\ [x : A]$*; then the following statements are equivalent:*

(1) There exists a decision function $\phi : A \to \mathsf{Boole}$ *such that, for all* $x \in A$, $\phi(x) =_{\mathsf{Boole}} \mathsf{true}$ *if and only if* $B(x)$ *is true.*

(2) for all $x \in A$, $B(x) \vee \neg B(x)$.

We can straight away prove that (1) implies (2). In fact, let us suppose that $\phi : A \to \mathsf{Boole}$ is a decision function for the proposition $B(x)\ \mathsf{prop}\ [x : A]$ and let us assume that $x \in A$. Then, because of corollary 4.2.5, we know that

$$(\phi(x) =_{\mathsf{Boole}} \mathsf{true}) \vee \neg(\phi(x) =_{\mathsf{Boole}} \mathsf{true})$$

and hence we can conclude

$$B(x) \vee \neg B(x)$$

by $\vee$-elimination. In fact $\phi(x) =_{\mathsf{Boole}} \mathsf{true}$ immediately implies that $B(x)$ is true, and hence that $B(x) \vee \neg B(x)$ is true, since ϕ is a decision function. On the other hand the same conclusion can be obtained from the assumption $\neg(\phi(x) =_{\mathsf{Boole}} \mathsf{true})$ by using the following derivation which again makes use of the fact that ϕ is a decision function:

$$\dfrac{\dfrac{[B(x)]_1}{\phi(x) =_{\mathsf{Boole}} \mathsf{true}} \quad \neg(\phi(x) =_{\mathsf{Boole}} \mathsf{true})}{\dfrac{\dfrac{\bot}{\neg B(x)}\ 1}{B(x) \vee \neg B(x)}}$$

Let us now show that (2) implies (1); we do not only have to provide a function $\phi : A \to \mathsf{Boole}$, which would be easy, but we have also to show that it is a decision function. This is the reason why we need some preliminary lemmas.

Lemma 4.2.8 *Let* $B(x)$ prop $[x : A]$*; then*

$$(\forall x \in A)\ B(x) \vee \neg B(x) \to \\ (\forall x \in A)(\exists y \in \mathsf{Boole})(y =_{\mathsf{Boole}} \mathsf{true} \to B(x))\ \&\ (y =_{\mathsf{Boole}} \mathsf{false} \to \neg B(x))$$

Proof. The proof is just an application of the $\vee$-elimination rule. In fact let us suppose that $(\forall x \in A)\ B(x) \vee \neg B(x)$ holds and assume that $x \in A$, then we have to show

$$(\exists y \in \mathsf{Boole})\ (y =_{\mathsf{Boole}} \mathsf{true} \to B(x))\ \&\ (y =_{\mathsf{Boole}} \mathsf{false} \to \neg B(x))$$

from $B(x) \vee \neg B(x)$ and hence all we need are the following deductions:

$$\dfrac{\dfrac{B(x)}{\mathsf{true} =_{\mathsf{Boole}} \mathsf{true} \to B(x)} \quad \dfrac{\neg(\mathsf{true} =_{\mathsf{Boole}} \mathsf{false})}{\mathsf{true} =_{\mathsf{Boole}} \mathsf{false} \to \neg B(x)}}{(\mathsf{true} =_{\mathsf{Boole}} \mathsf{true} \to B(x))\ \&\ (\mathsf{true} =_{\mathsf{Boole}} \mathsf{false} \to \neg B(x))} \\ \hline (\exists y \in \mathsf{Boole})\ (y =_{\mathsf{Boole}} \mathsf{true} \to B(x))\ \&\ (y =_{\mathsf{Boole}} \mathsf{false} \to \neg B(x))$$

and

$$\dfrac{\dfrac{\neg(\mathsf{false} =_{\mathsf{Boole}} \mathsf{true})}{\mathsf{false} =_{\mathsf{Boole}} \mathsf{true} \to B(x)} \quad \dfrac{\neg B(x)}{\mathsf{false} =_{\mathsf{Boole}} \mathsf{false} \to \neg B(x)}}{(\mathsf{false} =_{\mathsf{Boole}} \mathsf{true} \to B(x))\ \&\ (\mathsf{false} =_{\mathsf{Boole}} \mathsf{false} \to \neg B(x))} \\ \hline (\exists y \in \mathsf{Boole})\ (y =_{\mathsf{Boole}} \mathsf{true} \to B(x))\ \&\ (y =_{\mathsf{Boole}} \mathsf{false} \to \neg B(x))$$

Thanks to lemma 4.2.8, we are in the position to take advantage of the axiom of choice in order to obtain the following corollary.

Corollary 4.2.9 *Let* $B(x)$ prop $[x : A]$*; then*

$$(\forall x \in A)\ B(x) \vee \neg B(x) \to \\ (\exists \phi \in A \to \mathsf{Boole})(\forall x \in A)(\phi(x) =_{\mathsf{Boole}} \mathsf{true} \to B(x)) \& (\phi(x) =_{\mathsf{Boole}} \mathsf{false} \to \neg B(x))$$

Hence we have obtained the proof of the main theorem; in fact this corollary shows that if we have a proof of $(\forall x \in A)\ B(x) \vee \neg B(x)$ then both

(†) there exists a function $\phi : A \to \mathsf{Boole}$, and since all the proofs were developed within ITT we can effectively construct it, and

(‡) such a function is a decision function for $B(x)$; in fact if $\phi(x) =_{\mathsf{Boole}} \mathsf{true}$ then $B(x)$ is true and, on the other hand, if $B(x)$ is true we can use the following derivation to show that $\phi(x) =_{\mathsf{Boole}} \mathsf{true}$:

$$\dfrac{\phi(x) =_{\mathsf{Boole}} \mathsf{true} \vee \phi(x) =_{\mathsf{Boole}} \mathsf{false} \qquad \dfrac{\dfrac{[\phi(x) =_{\mathsf{Boole}} \mathsf{false}]_1 \quad \phi(x) =_{\mathsf{Boole}} \mathsf{false} \to \neg B(x)}{\neg B(x)} \qquad B(x)}{\dfrac{\bot}{\neg(\phi(x) =_{\mathsf{Boole}} \mathsf{false})}\ 1}}{\phi(x) =_{\mathsf{Boole}} \mathsf{true}}$$

We can save a curious reader the trouble of doing some work if we say that, supposing $h \in (\forall x \in A)\ B(x) \vee \neg B(x)$, the decision function which we obtain (after some unessential simplification) is

$$\begin{array}{lll} \phi \equiv & \lambda x.\ \mathsf{fst}(\mathsf{D}(& h(x), \\ & & (z)\ < \mathsf{true}, < \lambda w.\ z, \lambda u.\ R_0(\mathsf{shift}(*)) >>, \\ & & (z)\ < \mathsf{false}, < \lambda u.\ R_0(\mathsf{shift}(*)), \lambda w.\ z >>)). \end{array}$$

We can simplify it by far if we disregard all the parts which do not have a computational content and which appear in ϕ only because of the way we obtained it; in fact the function

$$\phi' \equiv \lambda x.\mathsf{D}(h(x), (z)\ \mathsf{true}, (z)\ \mathsf{false})$$

has obviously the same computational behavior as ϕ; the drawback is that we lack a *formal* proof that ϕ' is a decision function for $B(x)$. Of course we can obtain such a proof by using the fact that ϕ *is* a decision function for $B(x)$. In fact we can prove that $(\forall x \in A)\ \phi(x) =_{\mathsf{Boole}} \phi'(x)$ is true, because in general, supposing A, B, C, D are types, $c \in A + B$, $d(z) \in C \times D\ [z : A]$, $e(z) \in C \times D\ [z : B]$, the equality proposition

$$\mathsf{fst}(\mathsf{D}(c, (z)\ d(z), (z)\ e(z)) =_C \mathsf{D}(c, (z)\ \mathsf{fst}(d(z)), (z)\ \mathsf{fst}(e(z)))$$

holds.

4.3 An intuitionistic Cantor's theorem

Introduction

An intuitionistic version of Cantor's theorem, which shows that there is no surjective function from the type of the natural numbers N into the type $\mathsf{N} \to \mathsf{N}$ of the functions from N into N, is proved within Martin-Löf's Intuitionistic Type Theory with the universe of the small types.

4.3.1 Cantor's theorem

In this section we show that within Martin-Löf's Intuitionistic Type Theory with the universe of the small types [Mar84, NPS90] (ITT for short in the following) a version of Cantor's theorem holds, which shows that there is no surjective function from the type of the natural numbers N into the type $\mathsf{N} \to \mathsf{N}$ of the functions from N into N. As the matter of fact a similar result can be stated for any not-empty type A such that there exists a function from A into A which has no fixed point, as is the case of the successor function for the type N. In order to express Cantor's theorem within ITT we need the intensional equality proposition: let A be a type and $a, c \in A$, then by $a =_A c$ we mean the intensional equality proposition for elements of type A (see the type denoted by Id in the appendix B or [NPS90]).

Theorem 4.3.1 (ITT Cantor's theorem) *Let* N *be the type of the natural numbers; then*

$$\neg(\exists f \in \mathsf{N} \to (\mathsf{N} \to \mathsf{N}))(\forall y \in \mathsf{N} \to \mathsf{N})(\exists x \in \mathsf{N})\ f(x) =_{\mathsf{N}\to\mathsf{N}} y$$

To prove this theorem some lemmas are useful. Indeed we need to obtain a contradiction from the assumption

$$(\exists f \in \mathsf{N} \to (\mathsf{N} \to \mathsf{N}))(\forall y \in \mathsf{N} \to \mathsf{N})(\exists x \in \mathsf{N})\ f(x) =_{\mathsf{N}\to\mathsf{N}} y$$

i.e. from the two assumptions $f \in \mathsf{N} \to (\mathsf{N} \to \mathsf{N})$ and $(\forall y \in \mathsf{N} \to \mathsf{N})(\exists x \in \mathsf{N})\ f(x) =_{\mathsf{N}\to\mathsf{N}} y$.

By using the basic idea of the classic proof of Cantor's theorem, from the first assumption we can prove $\lambda x.\mathsf{s}(f(x)(x)) \in \mathsf{N} \to \mathsf{N}$, where $s : (x : \mathsf{N})\mathsf{N}$ is the successor function, by the following deduction:

$$\dfrac{\dfrac{\dfrac{[x : \mathsf{N}]_1 \qquad \dfrac{[x : \mathsf{N}]_1 \quad f \in \mathsf{N} \to (\mathsf{N} \to \mathsf{N})}{f(x) : \mathsf{N} \to \mathsf{N}}}{f(x)(x) \in \mathsf{N}}}{\mathsf{s}(f(x)(x)) \in \mathsf{N}}}{\lambda x.\mathsf{s}(f(x)(x)) \in \mathsf{N} \to \mathsf{N}}\ 1$$

We can now use this function in the second assumption in order to obtain

$$(\exists x \in \mathsf{N})\ f(x) =_{\mathsf{N}\to\mathsf{N}} \lambda x.\mathsf{s}(f(x)(x)).$$

So our problem becomes to obtain a contradiction from the two assumptions $x : \mathsf{N}$ and $f(x) =_{\mathsf{N}\to\mathsf{N}} \lambda x.\mathsf{s}(f(x)(x))$. We can use these assumptions to prove, by transitivity of the equality

proposition, that $f(x)(x) =_{\mathsf{N}} \mathsf{s}(f(x)(x))$ is true since in general if A and B are types and $a =_A c$ and $f =_{A\to B} g$ then $f(a) =_B g(c)$ and obviously $(\lambda x.\mathsf{s}(f(x)(x)))(x) =_{\mathsf{N}} \mathsf{s}(f(x)(x))$ is true.

We can thus re-state our aim by saying that we have to prove that ITT is not consistent with the assumption that the successor function has a fixed point. To prove this result we can transpose a well known categorical arguments within ITT [Law69, HP90]. Let us recall that we can solve the usual recursive definition of the sum between two natural numbers

$$\begin{cases} n + 0 = n : \mathsf{N} \\ n + \mathsf{s}(x) = \mathsf{s}(n + x) : \mathsf{N} \end{cases}$$

by putting $n + x \equiv \mathsf{N}_{rec}(x, n, (u, v)\ \mathsf{s}(v))$. Then the following lemma can be proved by induction.

Lemma 4.3.2 *For any* $n, x \in \mathsf{N}, n + \mathsf{s}(x) =_{\mathsf{N}} \mathsf{s}(n) + x$.

As for the sum, we can solve the recursive equation for the predecessor function

$$\begin{cases} p(0) = 0 : \mathsf{N} \\ p(\mathsf{s}(x)) = x : \mathsf{N} \end{cases}$$

by putting $p(x) \equiv \mathsf{N}_{rec}(x, 0, (u, v)\ u)$, and then that for the subtraction

$$\begin{cases} n - 0 = n : \mathsf{N} \\ n - \mathsf{s}(x) = p(n - x) : \mathsf{N} \end{cases}$$

by putting $n - x \equiv \mathsf{N}_{rec}(x, n, (u, v)\ p(v))$.

Lemma 4.3.3 *For any* $x \in \mathsf{N},\ (\forall n \in \mathsf{N})\ (n + x) - x =_{\mathsf{N}} n$.

Proof. By induction on x. If $x = 0$ then $(n + 0) - 0 =_{\mathsf{N}} n + 0 =_{\mathsf{N}} n$. On the other hand, if we assume by inductive hypothesis that $(\forall n \in \mathsf{N})\ (n + x) - x =_{\mathsf{N}} n$, then we obtain that $(n + \mathsf{s}(x)) - \mathsf{s}(x) =_{\mathsf{N}} p((n + \mathsf{s}(x)) - x) =_{\mathsf{N}} p((\mathsf{s}(n) + x) - x) =_{\mathsf{N}} p(\mathsf{s}(n)) =_{\mathsf{N}} n$. □

We can apply this lemma to the case $n = 0$ and obtain the following corollary.

Corollary 4.3.4 *For any* $x \in \mathsf{N},\ x - x =_{\mathsf{N}} 0$.

Proof. Immediate, since $0 + x =_{\mathsf{N}} x$ holds for each $x \in \mathsf{N}$. □

Now we conclude our proof. Let us write ω to mean the fixed point of the successor function, i.e. $\omega =_{\mathsf{N}} \mathsf{s}(\omega)$. Then the following lemma holds.

Lemma 4.3.5 *For any* $x \in \mathsf{N},\ \omega - x =_{\mathsf{N}} \omega$.

Proof. Again a proof by induction on x. If $x = 0$ then $\omega - 0 =_{\mathsf{N}} \omega$ and, supposing $\omega - x =_{\mathsf{N}} \omega$, we obtain $\omega - \mathsf{s}(x) =_{\mathsf{N}} p(\omega - x) =_{\mathsf{N}} p(\omega) =_{\mathsf{N}} p(\mathsf{s}(\omega)) =_{\mathsf{N}} \omega$. □

So we proved that $\omega - \omega =_{\mathsf{N}} 0$ by corollary 4.3.4 and also that $\omega - \omega =_{\mathsf{N}} \omega$ by lemma 4.3.5; hence $0 =_{\mathsf{N}} \omega =_{\mathsf{N}} \mathsf{s}(\omega)$. Finally we reach a contradiction.

Theorem 4.3.6 *For any* $x \in \mathsf{N},\ \neg(0 =_{\mathsf{N}} \mathsf{s}(x))$

Proof. By using an elimination rule for the type N, from the assumption $y : \mathsf{N}$, we obtain $\mathsf{N}_{rec}(y, \bot, (u, v)\ \top) \in \mathsf{U}$, where U is the universe of the small types, $\bot$ is the empty type and $\top$ is the one-element type. Now let us assume that $x \in \mathsf{N}$ and that $0 =_{\mathsf{N}} \mathsf{s}(x)$ is true, then $\mathsf{N}_{rec}(0, \bot, (u, v)\ \top) =_{\mathsf{U}} \mathsf{N}_{rec}(\mathsf{s}(x), \bot, (u, v)\ \top)$ since in general if A and B are types and $a =_A c$ is true and $b(x) \in B\ [x : A]$ then $b(a) =_B b(c)$ is true. Hence, by transitivity of the equality proposition, $\bot =_{\mathsf{U}} \top$ since $\bot =_{\mathsf{U}} \mathsf{N}_{rec}(0, \bot, (u, v)\ \top)$ and $\mathsf{N}_{rec}(\mathsf{s}(x), \bot, (u, v)\ \top) =_{\mathsf{U}} \top$. Then, because of one of the properties of the equality proposition for the elements of the type U, $\bot$ is inhabited since $\top$ is and hence, by discharging the assumption $0 =_{\mathsf{N}} \mathsf{s}(x)$, we obtain that $\neg(0 =_{\mathsf{N}} \mathsf{s}(x))$ is true. □

Thus the proof of theorem 4.3.1 is finished since we have obtained the contradiction we were looking for. Anyhow we stress on the fact that a similar result holds for any type A such that there exists a function from A into A with no fixed point. In fact, in this hypothesis, we can prove that there exists a function g from $A \to (A \to A)$ into $A \to A$ which supplies, for any function h from A into $A \to A$, a function $g(h) \in A \to A$ which is not in the image of h.

Theorem 4.3.7 *Let A be a type; then*

$$(\exists f \in A \to A)(\forall x \in A)\ \neg(f(x) =_A x) \to$$
$$(\exists g \in (A \to (A \to A)) \to (A \to A))(\forall h \in A \to (A \to A))(\forall x \in A)\ \neg(g(h) =_{A \to A} h(x))$$

The proof of this theorem is similar to the first part of the proof of theorem 4.3.1. In fact we only have to use the function $f \in A \to A$, instead of the successor function, to construct the function

$$g \equiv \lambda k.\lambda y.f(k(y)(y)) \in (A \to (A \to A)) \to (A \to A)$$

such that, for any $h \in A \to (A \to A)$ and any $x \in A$, allows to prove $h(x)(x) =_A f(h(x)(x))$, which is contrary to the assumption that the function f has no fixed point.

4.4 The forget-restore principle

Introduction

The aim of this section is to give a simple but instructive example of the forget-restore principle, conceived by Giovanni Sambin as a discipline for a constructive development of mathematics and first appeared in print in the introduction of [SV98]. The best way to explain such a philosophical position is to quote from that paper: "*To build up an abstract concept from a raw flow of data, one must disregard inessential details ... this is obtained by forgetting some information. To forget information is the same as to destroy something, in particular if there is no possibility of restoring that information ... our principle is that an abstraction is constructive ... when information ... is forgotten in such a way that it can be restored at will in any moment.*"

The example we want to show here refers to Martin-Löf's intuitionistic type theory (just *type theory* from now on). We assume knowledge of the main peculiarities of type theory, as formulated in [Mar84] or [NPS90].

Type theory is a logical calculus which adopts those notions and rules which keep *total* control of the amount of information contained in the different forms of judgment. However, type theory offers a way of "forgetting" information, that is, supposing A set, the form of judgment A true.

The meaning of A true is that there exists an element a such that $a \in A$ but it does not matter which particular element a is (see also the notion of *proof irrelevance* in [Bru80]). Thus to pass from the judgment $a \in A$ to the judgment A true is a clear example of the forgetting process.

We will show that it is a constructive way to forget since, provided that there is a proof of the judgment A true, an element a such that $a \in A$ can be re-constructed.

Of course the simple solution of adding *only* the rule

$$\frac{a \in A}{A\ \mathsf{true}}$$

allows to obtain such a result but is completely useless in practice. In fact, it does not allow to *operate* with judgments of the form A true and, in our experience, judgments of this form are essential in developing constructive mathematics, like for instance in formal topology, and in developing meta-mathematical arguments (see for instance [SV98] and [MV99]).

To obtain the same result, but avoiding this limitation, we provide a general calculus for expressions, directly inspired by Martin-Löf's Siena lectures in April 1983 (see [BV89]). This calculus was first published in [Val96a] and is similar for instance to that in [NPS90]. The advantage of our calculus with respect to the other ones present in the literature is that its rules, besides to allow to express all of the judgments of basic type theory, also permit a rigorous treatment of judgments of the form A true.

4.4.1 The multi-level typed lambda calculus

The first idea for the definition of our calculus is to use a sort of simple typed λ-calculus (see [Bar92]). In this way it is possible both to abstract on variables and to preserve a decidable theory of equality, which is an essential feature to describe any logical system since decidability is necessary in order to recognize the correct application of the inference rules. On the other hand, to describe type theory a simple typed λ-calculus is not sufficient. Thus we define the following multi-level typed λ-calculus: the intuitive idea is to construct a *tower* of dependent typed λ-calculi, each one over another, marked by a *level*. Hence the rules of the multi-level typed λ-calculus are those of a simple typed λ-calculus enriched by a label which specifies the level.

$$\text{(assumption)} \qquad \frac{\Gamma \vdash N :_i M}{\Gamma, x :_{i-1} N \vdash x :_{i-1} N} \qquad i \geq 1$$

$$\text{(weakening)} \qquad \frac{\Gamma \vdash N :_i M \qquad \Gamma \vdash K :_j L}{\Gamma, x :_{i-1} N \vdash K :_j L} \qquad i \geq 1$$

$$\text{(abstraction)} \qquad \frac{\Gamma, x :_j N \vdash K :_i L}{\Gamma \vdash ((x :_j N)K) :_i ((x :_j N)L)}$$

$$\text{(application)} \qquad \frac{\Gamma \vdash N :_i ((x :_j L)M) \qquad \Gamma \vdash K :_j L}{\Gamma \vdash N(K) :_i M[x := K]}$$

The assumption rule tells that, if N is an expression of level greater than zero, then we may assume it to be inhabited. The weakening rule tells that we may add assumptions of the form $x :_{i-1} N$ provided that the level of N is greater than zero. Abstraction and application are like usual, except that they apply to any level; note that they do not change the level of an expression.

These rules by themselves are not sufficient to develop any logical calculus since no expression can be assigned a type because to prove the conclusion of a rule one should have already proved its premise(s). So, in order to start, one needs some axioms. The first thing one has to do is to settle the maximum level m needed to describe a particular theory; to this aim we will introduce the symbol $*$ to indicate the only type of the highest level. One can then define all the other types downward from $*$ by means of axioms of the form $\vdash c :_m *$ for some constant c. Note that the only elements of $*$ are constants. Then, all the other axioms will have the form $\Gamma \vdash c :_{j-1} M$ for some constant c provided that $j > 0$ and there exists a type N such that $\Gamma \vdash M :_j N$. It is not difficult to recognize here some analogies with the approach to typed lambda-calculi used in the Pure Type Systems approach (see [Bar92]).

In the case of type theory, we define a chain

$$a :_0 A :_1 \mathsf{set} :_2 *$$

to mean that a is an element of A which is a set, i.e. an element of set, which is the only element of $*$. Thus our first axiom is:

$$\vdash \mathsf{set} :_2 *$$

We can now begin our description of type theory; to this aim we will follow the informal explanation by Martin-Löf in [Mar84]. We start by stating an axiom which introduces a constant for each set-constructor in correspondence with each formation rule of type theory. For instance, suppose we want to describe the set $\Pi(A, B)$; to this aim we add the axiom

$$\vdash \Pi :_1 (X :_1 \mathsf{set})(Y :_1 (x :_0 X)\ \mathsf{set})\ \mathsf{set}$$

which means that Π is a set-constructor constant which gives a new set when applied to the set X and to the propositional function Y on elements of X. It is straightforward to verify that this is a correct axiom since

$$\vdash (X :_1 \mathsf{set})(Y :_1 (x :_0 X)\ \mathsf{set})\ \mathsf{set} :_2 (X :_1 \mathsf{set})(Y :_1 (x :_0 X)\ \mathsf{set}) *$$

The next step corresponds to the introduction rule(s): we add a new axiom for each kind of canonical element. Let us consider again the case of the set $\Pi(A, B)$; then we put

$$\vdash \lambda :_0 (X :_1 \mathsf{set})(Y :_1 (x :_0 X)\ \mathsf{set})(y :_0 (x :_0 X)\ Y(x))\ \Pi(X, Y)$$

which states that, if X is a set, Y is a propositional function on elements of X and y is a function which gives a proof of $Y(x)$ for any x in X, then $\lambda(X,Y,y)$ is an element of the set $\Pi(X,Y)$. Thus this axiom is just a rephrasing of the Π-introduction rule in [Mar84].

Also the elimination rule becomes a new axiom. In fact, it defines the term-constructor constant introduced by the elimination rule. For instance for the set $\Pi(A,B)$, following the standard elimination rule (see the introduction of [Mar84]), we put

$$\begin{array}{l}\vdash F :_0 (X :_1 \mathsf{set})(Y :_1 (x :_0 X)\ \mathsf{set})(Z :_1 (z :_0 \Pi(X,Y))\ \mathsf{set}) \\ \qquad (c :_0 \Pi(X,Y))(d :_0 (y :_0 (x :_0 X)\ Y(x))\ Z(\lambda(X,Y,y)))\ Z(c)\end{array}$$

which states that, if X is a set, Y is a propositional function on elements of X, Z is a propositional function on elements of $\Pi(X,Y)$, c is an element of $\Pi(X,Y)$ and d is a method which maps any function y from x in X to $Y(x)$ into an element of $Z(\lambda(X,Y,y))$, then $F(X,Y,c,d)$ is an element of $Z(c)$.

In a similar way all of the rules of type theory become axioms of the multi-level typed λ-calculus.

The notion of level will not be necessary to prove the main theorem in this section but it is useful to prove that the multi-level typed lambda-calculus is normalizing. In fact, because of the presence of the levels, the multi-level typed lambda-calculus is obtained just putting together many dependent typed lambda-calculi with constants which cannot interact one with the other. Hence one can adapt to this framework any normalization proof for a dependent typed lambda calculus present in the literature (for instance see [CV98]). Anyway, in order to simplify the notation, in the following we will not write all the indexes of the levels whenever they are not necessary.

4.4.2 The judgment A true

The main novelty of our approach with respect to a standard simple typed lambda calculus, besides the notion of level, is that, besides the judgments of the form $N : M$ together with their rules, we can introduce here also the new form of judgment "M true", whose intended meaning is that the type M is inhabited. The rules we require on this form of judgment are completely similar to those for the judgment $N : M$ in the previous section. This fact will be crucial in the proof of the next theorem 4.4.1 which links the judgments of the form $N : M$ with those of the form M true.

$$\text{(assumption)}\quad \frac{\Gamma \vdash N :_i M}{\Gamma, x :_{i-1} N \vdash N\ \mathsf{true}} \qquad i \geq 1$$

$$\text{(weakening)}\quad \frac{\Gamma \vdash N :_i M \qquad \Gamma \vdash L\ \mathsf{true}}{\Gamma, x :_{i-1} N \vdash L\ \mathsf{true}} \qquad i \geq 1$$

$$\text{(abstraction)}\quad \frac{\Gamma, x :_j N \vdash L\ \mathsf{true}}{\Gamma \vdash ((x :_j N)L)\ \mathsf{true}}$$

$$\text{(application)}\quad \frac{\Gamma \vdash ((x :_j L)M)\ \mathsf{true} \qquad \Gamma \vdash K :_j L}{\Gamma \vdash M[x := K]\ \mathsf{true}}$$

It may be useful to note that in most of the previous rules, besides judgments of the form M true, it is necessary to use also those of the form $N : M$ and thus this calculus cannot be expressed independently from the previous one.

Like for the judgments of the form $N : M$ in the previous section, no type M can be proved to be inhabited, i.e. $\vdash M$ true, unless some specific axioms are added. The intended meaning of the judgment M true suggests to add the axiom $\Gamma \vdash M$ true whenever an axiom of the form $\Gamma \vdash c : M$ is present for some constant c. For instance, when we consider the set $\Pi(A,B)$ we will add the following two axioms:

$$\vdash (X : \mathsf{set})(Y : (x : X)\ \mathsf{set})(y : (x : X)\ Y(x))\ \Pi(X,Y)\ \mathsf{true}$$

which states that the type $(X : \mathsf{set})(Y : (x : X)\ \mathsf{set})(y : (x : X)\ Y(x))\ \Pi(X,Y)$ is inhabited; by using it, one can prove for instance that $\Gamma \vdash \Pi(A,B)$ true, provided that $\Gamma \vdash A : \mathsf{set}$ and $\Gamma, x : A \vdash B(x) : \mathsf{set}$ and $\Gamma, x : A \vdash B(x)$ true hold, since if $\Gamma, x : A \vdash B(x)$ true holds then, by the next theorem 4.4.1, it is possible to construct an expression b such that $\Gamma, x : A \vdash b(x) : B(x)$;

$$\begin{array}{ll}\vdash & (X : \mathsf{set})(Y : (x : X)\ \mathsf{set})(Z : (z : \Pi(X,Y))\ \mathsf{set}) \\ & (c : \Pi(X,Y))(d : (y : (x : X)\ Y(x))\ Z(\lambda(X,Y,y)))\ Z(c)\ \mathsf{true}\end{array}$$

which shows

$$\Gamma \vdash C(c)\ \mathsf{true}$$

provided that $\Gamma \vdash A : \mathsf{set}$, $\Gamma, x : A \vdash B(x) : \mathsf{set}$, $\Gamma, z : \Pi(A, B) \vdash C(z) : \mathsf{set}$, $\Gamma \vdash c : \Pi(A, B)$ and $\Gamma, y : (x : A)\ B(x) \vdash C(\lambda(A, B, y))\ \mathsf{true}$ hold. Note that, if the set $C(z)$ does not depend on z, the last axiom can be simplified to obtain

$$\Gamma \vdash C\ \mathsf{true}$$

provided that $\Gamma \vdash A : \mathsf{set}$, $\Gamma, x : A \vdash B(x) : \mathsf{set}$, $\Gamma, z : \Pi(A, B) \vdash C : \mathsf{set}$, $\Gamma \vdash \Pi(A, B)\ \mathsf{true}$ and $\Gamma, y : (x : A)\ B(x) \vdash C\ \mathsf{true}$ hold, since, by theorem 4.4.1, $\Gamma \vdash \Pi(A, B)\ \mathsf{true}$ implies that there exists an element c such that $\Gamma \vdash c : \Pi(A, B)$.

Since the rules for the judgment $N : M$ are completely similar to those for the judgment $M\ \mathsf{true}$ and whenever an axiom of the form $\Gamma \vdash c : M$ is added to the calculus also the axiom $\Gamma \vdash M\ \mathsf{true}$ is added, we can prove the following theorem 4.4.1. It allows to give a formal counterpart of the intended meaning of the judgment $\Gamma \vdash M\ \mathsf{true}$. Its proof, in one direction, shows how the reconstruct a witness for the judgment $M\ \mathsf{true}$ while, in the other, it shows how it is possible to forget safely.

Theorem 4.4.1 *Let Σ be any set of axioms of the form $\Gamma \vdash c : K$, for some constant c and type K, and let Σ^* be obtained from Σ by suitably substituting some of the axioms $\Gamma \vdash c : K$ in Σ with the corresponding axiom $\Gamma \vdash K\ \mathsf{true}$. Then $\Gamma \vdash M\ \mathsf{true}$ is derivable from Σ^* if and only if there exists an expression N such that $\Gamma \vdash N : M$ is derivable from Σ.*

Proof. In both directions the proof is by induction on the given proof. When we are "forgetting" we must start from below so that we know what can be forgotten. Let us show the inductive steps (provided Π is a proof, we will write Π^* to mean the proof obtained by inductive hypothesis).
(axiom)

$$\frac{\begin{array}{c}\Pi\\ \Gamma \vdash K :_i N\end{array}}{\Gamma \vdash c :_{i-1} K} \quad \Rightarrow \quad \frac{\begin{array}{c}\Pi\\ \Gamma \vdash K :_i N\end{array}}{\Gamma \vdash K\ \mathsf{true}} \quad i > 0$$

(assumption)

$$\frac{\begin{array}{c}\Pi\\ \Gamma \vdash N :_i M\end{array}}{\Gamma, x :_{i-1} N \vdash x :_{i-1} N} \quad \Rightarrow \quad \frac{\begin{array}{c}\Pi\\ \Gamma \vdash N :_i M\end{array}}{\Gamma, x :_{i-1} N \vdash N\ \mathsf{true}} \quad i > 0$$

(weakening)

$$\frac{\begin{array}{cc}\Pi & \Sigma\\ \Gamma \vdash N :_i M & \Gamma \vdash K :_j L\end{array}}{\Gamma, x :_{i-1} N \vdash K :_j L} \quad \Rightarrow \quad \frac{\begin{array}{cc}\Pi & \Sigma^*\\ \Gamma \vdash N :_i M & \Gamma \vdash L\ \mathsf{true}\end{array}}{\Gamma, x :_{i-1} N \vdash L\ \mathsf{true}} \quad i > 0$$

(abstraction)

$$\frac{\begin{array}{c}\Pi\\ \Gamma, x :_i N \vdash K :_j L\end{array}}{\Gamma \vdash ((x :_i N)K) :_j ((x :_i N)L)} \quad \Rightarrow \quad \frac{\begin{array}{c}\Pi^*\\ \Gamma, x :_i N \vdash L\ \mathsf{true}\end{array}}{\Gamma \vdash ((x :_i N)L)\ \mathsf{true}}$$

(application)

$$\frac{\begin{array}{cc}\Pi & \Sigma\\ \Gamma \vdash N :_j ((x :_i L)M) & \Gamma \vdash K :_i L\end{array}}{\Gamma \vdash N(K) :_j M[x := K]} \Rightarrow \frac{\begin{array}{cc}\Pi^* & \Sigma\\ \Gamma \vdash ((x :_i L)M)\ \mathsf{true} & \Gamma \vdash K :_i L\end{array}}{\Gamma \vdash M[x := K]\ \mathsf{true}}$$

It should now be clear how we obtain the set of axioms Σ^* from the set of axioms Σ: we have to change only those axioms which appear on a modified proof and this is the reason why we have to "forget" from below: for instance in the rules of weakening or application only one the premises is modified and only the axioms on that premise have to be changed.

On the other side, in the case we are "restoring", we must start from above in such a way that an axiom (possibly in Σ^*) is replaced with a suitable instance of an axiom (in Σ).

(axiom)

$$\frac{\begin{matrix}\Pi\\ \Gamma \vdash M :_i N\end{matrix}}{\Gamma \vdash M \ \mathsf{true}} \quad \Rightarrow \quad \frac{\begin{matrix}\Pi\\ \Gamma \vdash M :_i N\end{matrix}}{\Gamma \vdash c :_{i-1} M} \quad i > 0$$

(assumption)

$$\frac{\begin{matrix}\Pi\\ \Gamma \vdash N :_i M\end{matrix}}{\Gamma, x :_{i-1} N \vdash N \ \mathsf{true}} \quad \Rightarrow \quad \frac{\begin{matrix}\Pi\\ \Gamma \vdash N :_i M\end{matrix}}{\Gamma, x :_{i-1} N \vdash x :_{i-1} N} \quad i > 0$$

(weakening)

$$\frac{\begin{matrix}\Pi & \Sigma\\ \Gamma \vdash N :_i M & \Gamma \vdash L \ \mathsf{true}\end{matrix}}{\Gamma, x :_{i-1} N \vdash L \ \mathsf{true}} \quad \Rightarrow \quad \frac{\begin{matrix}\Pi & \Sigma^*\\ \Gamma \vdash N :_i M & \Gamma \vdash K :_j L\end{matrix}}{\Gamma, x :_{i-1} N \vdash K :_j L} \quad i > 0$$

(abstraction)

$$\frac{\begin{matrix}\Pi\\ \Gamma, x :_i N \vdash L \ \mathsf{true}\end{matrix}}{\Gamma \vdash ((x :_i N)L) \ \mathsf{true}} \quad \Rightarrow \quad \frac{\begin{matrix}\Pi^*\\ \Gamma, x :_i N \vdash K :_j L\end{matrix}}{\Gamma \vdash ((x :_i N)K) :_j ((x :_i N)L)}$$

(application)

$$\frac{\begin{matrix}\Pi & \Sigma\\ \Gamma \vdash ((x :_i L)M) \ \mathsf{true} & \Gamma \vdash K :_i L\end{matrix}}{\Gamma \vdash M[x := K] \ \mathsf{true}} \Rightarrow \frac{\begin{matrix}\Pi^* & \Sigma\\ \Gamma \vdash N :_j ((x :_i L)M) & \Gamma \vdash K :_i L\end{matrix}}{\Gamma \vdash N(K) :_j M[x := K]}$$

It is worth noting that in the process which transforms first the proof of $\Gamma \vdash N : M$ into $\Gamma \vdash M \ \mathsf{true}$ and then into $\Gamma \vdash N' : M$ we will not in general obtain the same element, i.e. N and N' may differ for the constants used in the axioms with the same type.

4.4.3 Final remarks

What we have illustrated in the previous sections is just an example of the process of "forgetting"; for instance, as somebody suggested after reading a first version of this section, one could consider also the judgments $M \ \mathsf{type}$ and $M \ \mathsf{element}$ as a forgetting abbreviation for the judgment $M :_j N$ with $j > 0$ and $j = 0$ respectively and develop for these judgments a suitable calculus analogous to the one we proposed for the judgment $M \ \mathsf{true}$.

Moreover it should be clear that what we have done is just a simple illustration of the forget-restore paradigm and that it is not a complete description of a full theory for judgments of the form $A \ \mathsf{true}$ within type theory. In fact we chose to develop a dependent type multilevel lambda calculus since it is well suited for the framework of the Martin-Löf's dependent type theory that we have described but it is not of immediate application if we consider also the non-dependent part of the theory like for instance when we define $A \to B$ as $\Pi(A, (x : A)\ B)$ provided that the proposition B does not depend on the elements of A. For instance the rule

$$\frac{\Gamma \vdash A \to B \ \mathsf{true} \qquad \Gamma \vdash A \ \mathsf{true}}{\Gamma \vdash B \ \mathsf{true}}$$

is admissible in our system but it is not derivable ; hence we have a too weak theory for judgments of the form $A \ \mathsf{true}$. To solve this problem the first step is to be able to deal also with *assumptions* of the form $A \ \mathsf{true}$, instead that only with those of the form $x : A$, when the variable x does not appear in the conclusion $B \ \mathsf{true}$. This is not possible in a general dependent type calculus since even a conclusion of the form $B \ \mathsf{true}$ may in general depend on the variables in the assumptions.

We can obtain this result if, when performing the forgetting process, we will take into account also which variables appear in the types in the conclusion of a rule. Thus we will have the following transformation of a *full* proof into a *forgetting* one:

$$\text{(assumption)} \quad \frac{\begin{matrix}\Pi\\ \Gamma \vdash N :_i M\end{matrix}}{\Gamma, x :_{i-1} N \vdash x :_{i-1} N} \quad \Rightarrow \quad \frac{\begin{matrix}\Pi\\ \Gamma \vdash N :_i M\end{matrix}}{\Gamma, N \ \mathsf{true} \vdash N \ \mathsf{true}}$$

since the variable x is introduced by the rule and hence cannot appear in N;

$$\text{(weakening)} \quad \frac{\begin{array}{cc}\Pi & \Sigma \\ \Gamma \vdash N :_i M & \Gamma \vdash K :_j L\end{array}}{\Gamma, x :_{i-1} N \vdash K :_j L} \quad \Rightarrow \quad \frac{\begin{array}{cc}\Pi & \Sigma^* \\ \Gamma \vdash N :_i M & \Gamma \vdash L\ \mathsf{true}\end{array}}{\Gamma, N\ \mathsf{true} \vdash L\ \mathsf{true}}$$

since the variable x is assumed by weakening and hence it cannot appear in L. The case of the abstraction rule

$$\text{(abstraction)} \quad \frac{\begin{array}{c}\Pi \\ \Gamma, x :_i N \vdash K :_j L\end{array}}{\Gamma \vdash ((x :_i N)K) :_j ((x :_i N)L)}$$

deserves a more detailed analysis; in fact we can surely use the transformation that we have proposed in the proof of theorem 4.4.1, but, provided the variable x does not appear in L, also the following transformation can be used

$$\frac{\begin{array}{c}\Pi^* \\ \Gamma, N\ \mathsf{true} \vdash L\ \mathsf{true}\end{array}}{\Gamma \vdash ((N)L)\ \mathsf{true}}$$

where we have introduced the new notation $((N)L)$ to mean that the abstracted variables does not appear in the body of the abstraction. Finally also for the application rule

$$\text{(application)} \quad \frac{\begin{array}{cc}\Pi & \Sigma \\ \Gamma \vdash N :_j ((x :_i L)M) & \Gamma \vdash K :_i L\end{array}}{\Gamma \vdash N(K) :_j M[x := K]}$$

two transformations are possible according to the variables which appear in the conclusion. The first is the one that we used in the proof of theorem 4.4.1 and it can be applied in any case. However, provided M does not depend on x, it is possible to use also the following

$$\frac{\begin{array}{cc}\Pi^* & \Sigma^* \\ \Gamma \vdash ((L)M)\ \mathsf{true} & \Gamma \vdash L\ \mathsf{true}\end{array}}{\Gamma \vdash M\ \mathsf{true}}$$

It is now possible to change also the form of the axioms. Here we will show only a simple example. Suppose that we want to introduce the type $A \to B$. Then we need the following axioms:

$$\begin{array}{l}\vdash \to :_1 (X :_1 \mathsf{set})(Y :_1 \mathsf{set})\ \mathsf{set} \\ \vdash \lambda :_0 (X :_1 \mathsf{set})(Y :_1 \mathsf{set})(y :_0 (x :_0 X)\ Y)\ X \to Y \\ \vdash \mathsf{Ap} :_0 (X :_1 \mathsf{set})(Y :_1 \mathsf{set})(f :_0 X \to Y)(x :_0 X)\ Y\end{array}$$

If we now consider the transformations used in the prove of theorem 4.4.1 we will obtain

$$\begin{array}{l}\vdash (X :_1 \mathsf{set})(Y :_1 \mathsf{set})(y :_0 (x :_0 X)\ Y)\ X \to Y\ \mathsf{true} \\ \vdash (X :_1 \mathsf{set})(Y :_1 \mathsf{set})(f :_0 X \to Y)(x :_0 X)\ Y\ \mathsf{true}\end{array}$$

but, provided that we use also the notation $((X)Y)$ for the abstractions when Y does not depends on the elements in X, we can add to them the following new axioms:

$$\begin{array}{l}\vdash (X :_1 \mathsf{set})(Y :_1 \mathsf{set})(((X)\ Y)\ X \to Y)\ \mathsf{true} \\ \vdash (X :_1 \mathsf{set})(Y :_1 \mathsf{set})(f :_0 X \to Y)((X)\ Y)\ \mathsf{true} \\ \vdash (X :_1 \mathsf{set})(Y :_1 \mathsf{set})((X \to Y)(x :_0 X)\ Y)\ \mathsf{true} \\ \vdash (X :_1 \mathsf{set})(Y :_1 \mathsf{set})((X \to Y)((X)\ Y))\ \mathsf{true}\end{array}$$

and it is straightforward to use the last one to derive the rule

$$\frac{\Gamma \vdash A \to B\ \mathsf{true} \qquad \Gamma \vdash A\ \mathsf{true}}{\Gamma \vdash B\ \mathsf{true}}$$

Since any of the new axioms is the result of a forgetting process from a standard axiom and we can restore it simply by adding the abstracted variables, which can be done in an algorithmic way, this is again an instance of a constructive way of forgetting and a theorem like theorem 4.4.1 can be proved also in this case.

Chapter 5

Set Theory

5.1 Summary

In this chapter we will show how a local set theory can be developed within Martin-Löf's type theory. In particular, we will show how to develop a predicative theory of subsets within type theory. In fact, a few years' experience in developing constructive topology in the framework of type theory has taught us that a more liberal treatment of subsets is needed than what could be achieved by remaining literally inside type theory and its traditional notation. To be able to work freely with subsets in the usual style of mathematics one must come to conceive them like any other mathematical object (which technically means for instance that the judgment that something is a subset can be taken as assumption) and have access to their usual apparatus (for instance, union and intersection).

Many approaches were proposed for the development of a set theory within Martin-Löf's type theory or in such a way that they can be interpreted in Martin-Löf's type theory (see for instance [NPS90], [Acz78], [Acz82], [Acz86], [Hof94], [Hof97] and [HS95]).

The main difference between our approach and these other ones stays in the fact that we do not require a subset to be a set while in general in the other approaches the aim is to apply the usual set-constructors of basic type theory to a wider notion of set, which includes sets obtained by comprehension over a given set (see [NPS90], [Hof94], [Hof97] and [HS95]) or to define an axiomatic set theory whose axioms have a constructive interpretation (see [Acz78], [Acz82], [Acz86]); hence the justification of the validity of the rules and axioms for sets must be given anew and a new justification must be furnished each time the basic type theory or the basic axioms are modified.

In this chapter we will provide all the main definitions for a subset theory; in particular, we will state the basic notion of subset U of a set S, that is we will identify U with a propositional function over S and hence we will require that a subset is never a set; then we will introduce the membership relation between an element a of S and a subset U of S, which will turn out to hold if and only if the proposition $U(a)$ is true. Then the full theory will be developed based on these simple ideas, that is the usual set-theoretic operations will be defined in terms of logical operations. Of course not all of the classical set theoretic axioms are satisfied in this approach. In particular it is not possible to validate the axioms which do not have a constructive explanation like, for instance, the power-set construction.

5.2 Introduction

The present work originates from the need of subsets in the practical development of a constructive approach to topology (see [Sam87] and [Sam97] and references therein) and most of what we present here is the answer to problems actually encountered. The solution we arrived at is inspired by a philosophical attitude, and this too has become clear along the way. Thus the aim of this Introduction is to explain such attitude and to make the resulting principles explicit, which might help to understand better some of the technical choices.

5.2.1 Foreword

Beginning in 1970, Per Martin-Löf has developed an intuitionistic type theory (henceforth type theory tout-court) as a constructive alternative to the usual foundation of mathematics based on classical set theory. We assume the reader is aware at least of the main peculiarities of type theory, as formulated in [Mar84] or [NPS90]; here we recall some of them to be able to introduce our point of view.

The form of type theory is that of a logical calculus, where inference rules to derive judgments are at the same time set theoretic constructions, because of the "propositions-as-sets"[1] interpretation. The spirit of type theory - expressing our interpretation in a single sentence - is to adopt those notions and rules which keep total control of the amount of information contained in the different forms of judgment. We now briefly justify this claim.

First of all, the judgment asserting the truth of a proposition A, which from an intuitionistic point of view means the existence of a verification of A, in type theory is replaced by the judgment $a \in A$ which explicitly exhibits a verification a of A. In fact, it would be unwise, for a constructivist, to throw away the specific verification of A which must be known to be able to assert the existence of a verification!

The judgment that A is a set, which from an intuitionistic point of view means that there exists an inductive presentation of A, is treated in type theory in a quite similar way (even if in this case no notation analogous to $a \in A$ is used) since the judgment A set in type theory becomes explicit knowledge of the specific inductive presentation of A. In fact, the rules for primitive types and for type constructors are so devised that whenever a judgment A set is proved, it means that one has also complete information on the rules which describe how canonical elements of A are formed. Such property, which might look like a peculiarity of type theory, is as a matter of fact necessary in order to give a coherent constructive treatment of quantifiers. Consider for instance universal quantification, and take for granted that an intuitionistically meaningful explanation of universal quantification is possible only for domains with an inductive presentation, that is for what have been called sets above (by the way, this is the reason why the distinction between sets and types is so basic in type theory, see [Mar84]). Then the pure knowledge that A is a set is sufficient to say that universal quantification over A gives rise to propositions; however, it would be unwise to forget which specific rules generate A inductively, since, in general, a method verifying a universally quantified proposition over A can be produced, by means of an elimination rule, only by direct reference to the method by which A is generated.

Summing up, we see not only that type theory is inspired by the principle of control of information, but also that the same principle should be at the base of any coherent treatment of sets and propositions, if it has to be both intuitionistic and free of wastes.

Coming back to the formalism in which type theory is expressed, one can see that the principle of keeping control of information is revealed also at the level of syntax, since most inference rules are formulated in a fully analytic style, i.e. everything which appears in the premises is present somehow also in the conclusion. A consequence is, for instance, that the derivation of a judgment $a \in A$ is so detailed that a is *ipso facto* a program which satisfies the requirements specified by A. This is why type theory is particularly interesting for computer science.

However, our experience in developing pieces of actual mathematics within type theory has led us to believe that "orthodox" type theory is not suitable because its control of information is too strict for this purpose. In fact, the fully analytic character of type theory becomes a burden when dealing with the synthetic methods of mathematics, which "forget" or take for granted most of the details. This, in our opinion, could be the reason why type theory has collected, up to now, more interest among logicians and computer scientists as a formal system than among mathematicians as a foundational theory.

We claim that there is no intrinsic reason why it should remain so, and that actually it is only a matter of developing a stock of "utilities", that is, of building up a toolbox which covers the territory between the basic formalism and mathematical practice; after all, this happened for classical set theory ZF long ago. In other words, the situation seems analogous to that of a programmer who, maybe because of a particular skill with the machine language, has not yet developed those higher

[1] Called "formulae-as-types" in [How80]. A little warning on terminology is necessary here: we use "set" exactly as in [Mar84], while "category" of [Mar84] is here replaced with "type"as in [NPS90].

level constructs and languages which allow him to save time and mental energy, and thus in the end are necessary to free the mind for a common human, i.e. abstract, comprehension.

So our general aim is to build up those tools, that is definitions and rules, which "forget" some information, and thus allow a higher level of abstraction, which can make type theory more handy and suitable to work out (intuitionistic) mathematics basing on mathematical intuition, as it has been and should be.

5.2.2 Contents

Here, completing the work first announced in [SV93], we make a substantial step in the direction stated above and show how to develop a predicative theory of subsets within type theory.[2] A few years' experience in developing topology in the framework of type theory has taught us that a more liberal treatment of subsets is needed than what could be achieved by remaining literally inside type theory and its traditional notation. In fact, to be able to work freely with subsets in the usual style of mathematics one must come to conceive them like any other mathematical object (which technically means for instance that the judgment that something is a subset can be taken as assumption) and have access to their usual apparatus (for instance, union and intersection).

Subset theory as developed in [NPS90] does not meet the above demands since, being motivated by programming, its aim is different. We could say, in fact, that the aim of [NPS90] is to apply the usual set-constructors of basic type theory to a wider notion of set, which includes sets obtained by comprehension over a given set; the price *they* have to pay is that the justification of the validity of rules for sets must be given anew. The same price must be paid whenever the basic type theory is modified; another example is the recent [Tur97].

The way out is to adopt the simple idea that it is not compulsory that a subset be a set. Then one is free to define subsets in a natural way as propositional functions (as first suggested in [Mar84], page 64, and explicitly adopted in [Sam87]), and then to introduce the new notion of element of a subset, in terms of which also the other standard notions, like inclusion and extensional equality, arbitrary unions and intersections, singletons and finite subsets, quantifiers and functions defined on subsets can be defined. The resulting subset theory is a sort of type-less set theory localized to a set and we have experienced that it is sufficient for instance for the development of topology. We prove that all of this can be done in type theory without losing control, that is by "forgetting" only information which can be restored at will. This is reduced to the single fact that, for any set A, the judgment A `true` holds if and only if there exists a such that $a \in A$, and it can be proved once and for all, see [Val98]; this is the price *we* have to pay to justify our approach.

Since for all notions related to subsets we adopt essentially the standard notation, the result is that at first sight a page of mathematics written using subset theory looks like a page of standard mathematical writing, and one might easily overlook or forget the underlying substantial novelty, namely that everything is directly formalized in type theory. Far from being a drawback, this is in a sense our main intention, since it would show that one can develop an intuition free from the underlying formalism.

5.2.3 Philosophical motivations

The attitude illustrated so far in this introduction can be seen as the specific outcome of a more general philosophical position (see [Sam91]) when applied to type theory, and the results we prove can be seen as fragments of a general program (see [PV93]) not necessarily bound to type theory. Here we describe briefly both the philosophical position and the general program in the form of some principles, just enough to be able to state the connection with the problem of foundations.

To build up an abstract concept from a raw flow of data, one must disregard inessential details; in other words, to simplify the complexity of concrete reality one must idealize over it, and this is obtained by "forgetting" some information. To forget information is the same as to destroy something, in particular if there is no possibility of restoring that information, like when the magnetic memory of a disk is erased. So to abstract involves a certain amount of destruction; our principle is that an abstraction is constructive, that is, a reliable tool in getting knowledge which

[2] The second step should be on quotient sets, or abstract data types, or setoids.

is faithful to reality, not when information is kept as much as possible, but when it is "forgotten" in such a way that it can be restored at will in any moment. This after all is the test to show that an abstraction does not lead astray from reality, that is, that it preserves truth.

It is clear that the first step, and often the only one, to be able to restore what has been "forgotten" is to know, to be aware of, what has been forgotten, and to keep control of it. So the second principle is that constructivism does not consist of a *a priori* self-limitation to full information, which would tie constructivism with reluctance to abstraction (as was the case around the twenties when finitism and intuitionism were somehow identified), but rather in the awareness of the destruction which has been operated to build up a certain abstract concept.

When it comes to mathematical terms, awareness of what has been destroyed or forgotten can sometimes be put in objective terms, and then it takes the form of a method by which the previous state can be restored. If T' is a theory obtained from a more basic theory T by adding some more abstract constructs and their rules, then a method must be supplied which allows us to transform whatever proof in T' into a proof within T *with the same computational content*. This allows us to "forget safely", since it guarantees that faithfulness to the more concrete level of T is not lost by the more abstract concepts of T'.

This, we believe, is the only reasonable way for a constructivist to extract a philosophical and mathematical value out of Hilbert's program. To obtain a foundation which justifies itself, in Hilbert's view it is necessary to rely on a part of mathematics, called real mathematics, which has to be safe beyond any doubt and without any proof. In Hilbert's conception this is identified with finitistic mathematics, that is manipulation of concrete objects; here instead real mathematics is identified with type theory, which is of course far richer than finitistic mathematics but still serves the purpose. In fact, on the one hand the contextual explanation of judgments and rules and on the other hand its interpretation as a programming language (the modern "manipulation of concrete objects") indeed make it reliable beyond doubt and without any proof.

Hilbert was right, of course, in saying that mathematics cannot be restricted to real mathematics; in fact, even the most radical constructivist certainly uses more abstract notions or ideas, even if they do not appear in his communications. But which abstract notions can be accepted? We propose an answer here. It is well known that Hilbert's view puts no limitation, as long as the consistency of the formalism in which ideal mathematics is expressed is proven within real mathematics. This cannot be accepted by a constructivist, since a consistency proof is not enough to restore, once and for all, the constructive meaning, i.e. faithfulness to the concrete reality of computations. After all, even classical logic is consistent, and with a finitistic proof! So the program is to analyze, case by case, how a certain abstract notion is linked with real mathematics; when it is clear which concrete aspects are forgotten and how they can be restored by a suitable method, then that abstract notion can be used freely and safely. In this chapter we show how this is possible for the theory of subsets, and thus we accomplish a fragment of the constructive version of Hilbert's program, which we have called the Camerino program from the place where we spoke about it for the first time (see [PV93]). The aim is, paradoxically, to save the intuition of an intuitionist from the rigidity of formal systems by supplying safe bridges between intuition and computation.

5.3 Reconstructing subset theory

5.3.1 The notion of subset

In classical mathematics a subset U of a set S is usually defined to be a set such that if $a \in U$ then $a \in S$. Importing in type theory this definition as it stands, however, would give a notion of subset not general enough to include all examples of what is undoubtedly to be considered a subset. In fact, if S is a set and $U : (x : S)\ \mathsf{prop}$ is a propositional function[3] over S, then we surely want the collection of elements of S satisfying U, usually written $\{x \in S |\ U(x)\}$, to be a subset of S. In ZF, $\{x \in S |\ U(x)\}$ would be a set, by the separation principle; in type theory, however, no form of the separation principle is justified, since in general there are no rules telling how the canonical elements of the collection $\{x \in S |\ U(x)\}$ are formed. In fact, a is an element of

[3]Which means that U applied to x, for $x \in S$, is a proposition, written $U(x)\ \mathsf{prop}\ [x : S]$.

$\{x \in S|\ U(x)\}$ if $a \in S$ and $U(a)$ is true, that is, if there exists b such that $b \in U(a)$, but this form of judgment is not one of the four forms of judgments considered in type theory and hence there is no canonical way to reach the conclusion that such b exists. For example, if $U(x)$ is the property "the Turing machine with index x does not stop on input x", then there are no effective rules to generate $\{x \in \mathsf{N}|\ U(x)\}$. Thus, the conclusion is that we want $\{x \in S|\ U(x)\}$ to be a subset of S for any property U, but also that it does not need to be a set.

A second observation is that in ordinary mathematics, operations like union and intersection are freely defined on the class of all sets, while at the opposite extreme in type theory there is no operation of union or intersection in the ordinary sense available on sets. In fact, the result of any operation of set-formation gives rise to a set whose elements are specific of the constructed set, and thus, for instance, we could not have a common statement like $a \in S \cap T$ iff $a \in S$ and $a \in T$, because if $\cap$ were a set-constructor then $S \cap T$ would be a set different from S and T, and hence its elements could not be in common with S and T. As we will soon see, however, such set-theoretic operations can easily be defined on subsets of a set, as soon as we do not require a subset to be a set. We are thus led to take the step of relaxing the requirement that a subset be a set. Therefore a subset will not have canonical elements, nor rules of elimination or of equality.

Two ways of defining subsets are traditionally available which do not require a subset to be a set. The first is that a subset of S is given by a property $U(x)$ with x ranging over S; while in a classical perspective it can be conceived that there are many more subsets than properties, from a constructive point of view there is no sense in assuming the existence of a subset unless we can specify it, namely by a property. Thus the conclusion would be that a subset U of S is nothing but a propositional function U over S.

The second conception of subset of S, namely as a function $f : S \to \{0, 1\}$ usually called characteristic function, brings in the end to the same conclusion, as we now see. Classically, any function $f : S \to \{0, 1\}$ gives rise to a property over S, namely the property $f(x) = 1$, and, given a property $U(x)$ over S, the associated function is

$$f_U(x) = \begin{cases} 1 & \text{if } U(x) \text{ true} \\ 0 & \text{otherwise} \end{cases}$$

If we transfer this as it stands into type theory, we obtain a notion of subset which is too narrow. In fact, due to the different notion of function, the above argument, when properly translated in type theory, gives a bijective correspondence between functions $S \to \{0, 1\}$ and *decidable* propositional functions over S (for a detailed proof, see for instance section 4.2 or [Val96]).

However, in the classical conception the above definition of f_U can be seen just as a different way of denoting the propositional function U itself. In fact, classically a proposition is just a way to denote a truth value (see [Fre1892]), so $\{0, 1\}$ can be identified with the set of values of propositions. Under this reading, the intuitionistic analogue of a characteristic function is a function from S into the type of intuitionistic propositions, i.e. a propositional function over S.

So both traditional approaches lead to the same intuitionistic version. We thus put:

Definition 5.3.1 (Definition of subset) *For any set S, a propositional function U with argument ranging in S is called a subset of S, and is written $U \subseteq S$.*

Thus we can think that a subset U of S is obtained by abstracting the variable x in the judgment $U(x)$ prop $[x : S]$, i.e. $U \equiv (x : S)\ U(x)$. The same effect is usually expressed with the brace notation to form a subset $\{x \in S|\ U(x)\}$, which does not depend on x any longer. So we put:

$$\{x \in S|\ U(x)\} \equiv U \equiv (x : S)\ U(x)$$

However, it must be said explicitly that, even if we adopt the common expression $\{x \in S|\ U(x)\}$ for a subset, it remains true that a subset is a propositional function and hence a subset can *never* coincide with a set, for the simple reason that propositional functions are of a type different from that of sets.

By similar reasons, the notion of subset is not automatically accompanied by that of element of a subset: writing $a \in \{x \in S|\ U(x)\}$, for $a \in S$, does never give a well formed expression and, on the other hand, writing $u : U$ would mean $(x : S)\ u(x) : (x : S)\ U(x)$, which corresponds to the

judgment $u(x) \in U(x)$ $[x : S]$ in the notation of [Mar84], and hence has nothing to do with the intuitive notion of element of the subset U. So this notion has to be introduced anew. And indeed we need it, because only in virtue of it an extensional theory of subsets can be reconstructed like that of usual mathematical practice; for instance, we surely want two subsets to be equal iff they have the same elements.[4]

It is worth noting that much of what we are going to do in the case of subsets extends to relations in a natural way. In fact, contrary to the classical approach, a relation in type theory is just a propositional function with several arguments and thus it is a straightforward generalization of the notion of subset.

5.3.2 Elements of a subset

Given a set S and a subset $U \subseteq S$, the intuitive idea is that the element a of S is an element of U when the property U holds on a. In type theory, this is expressed by requiring $U(a)$ true, which means that there exists b such that $b \in U(a)$. However, as in mathematical practice, we surely wish not to bother about the information of the specific b which makes $U(a)$ true: for a to be an element of U, it is the pure existence of a proof which is required and not the actual specific verification, which we want to "forget"[5]. Theorem 4.4.1 in chapter 4 tells that we can restore such information when wished, at the cost of some meta-mathematical work.

In the same time, it is essential to keep the information of which element a is (see for instance $\subseteq_S$-elimination in proposition 5.3.4), and thus express "U holds on a" rather than "$U(a)$ true". In fact, $U(a)$ may loose the information of which element a is considered without the possibility of restoring it from $U(a)$ true. For instance, if $U \equiv (x : S)\ \mathsf{N}$, where N is the set of natural numbers, then $U(a) \equiv ((x : S)\ \mathsf{N})(a) = \mathsf{N}$ is true, but there is no way to recover the element a to which U is applied.

Therefore, what we wish is a proposition $a\ \epsilon_S\ U$ which, besides giving $U(a)$ true, "recalls" which a is considered, that is, which satisfies

$$(*)\ a\ \epsilon_S\ U\ \mathsf{true} \text{ iff } U(a)\ \mathsf{true} \text{ and } a \in S$$

Note that the right side of $(*)$ is the conjunction of two judgments, which is usually not treated in type theory: this is the problem we have to face.

It can be shown that $(*)$ is equivalent to the following two conditions together

- for every $a \in S$, $a\ \epsilon_S\ U$ true iff $U(a)$ true
- if $a\ \epsilon_S\ U$ true, then $a \in S$

To develop subset theory more smoothly, however, it is convenient to adopt an apparently stronger formulation in which the first condition is expressed by a proposition, namely the following conditions:

1. $(\forall x \in S)\ (x\ \epsilon_S\ U \leftrightarrow U(x))$ true
2. if $a\ \epsilon_S\ U$ true, then $a \in S$

From now on, we will refer to them as the first and second ϵ-condition; we will see that they are all what is needed to be able to develop all of subset theory.

Now, to solve the ϵ-conditions, that is to find a proposition which satisfies them, the crucial remark is that there is substantially one way to include the information given by the judgment $a \in S$ into a proposition, and that is $\mathsf{Id}(S, a, a)$. In fact, it is easy to prove that $a \in S$ if and only if $\mathsf{Id}(S, a, a)$ true: one direction is just the rule of Id-introduction, while the other is obtained by a simple meta-mathematical argument, namely that from a proof of $\mathsf{Id}(S, a, a)$ true one can effectively

[4] While the identification of subsets with propositional functions is common to several approaches (for instance see [Coq90] for a calculus of constructions), an explicit introduction of the notion of element of a subset seems to be peculiar of the present one. The details to export it to other type theories must be worked out on purpose.

[5] After the talk in Venice during the conference "Twenty-five years of Constructive Type Theory" in October 1995, Prof. de Bruijn has kindly called our attention to his notion of *proof-irrelevance* (see [Bru80]), which seems connected with our idea of "forgetting".

obtain a proof of $\mathsf{Id}(S, a, a)$ prop, which in turn must include a proof of $a \in S$. This is the only addition to be made on top of an implementation of type theory to obtain an implementation of our toolbox. Note that requiring a *formal* equivalence would not make sense.

Thus we simply put

$$x \,\epsilon_S\, U \equiv\ U(x)\ \&\ \mathsf{Id}(S, x, x)$$

The verification of the ϵ-conditions is immediate; let us note explicitly, however, that to prove $U(x)\ \&\ \mathsf{Id}(S, x, x) \leftrightarrow U(x)\ \mathsf{true}$ the knowledge of $x \in S$ is essential. This agrees perfectly with the informal requirement that the proposition $a\ \epsilon_S\ U$ must coincide with $U(a)$ when $a \in S$ is known, but differs from $U(a)$ since it keeps track of a by containing knowledge of $a \in S$.

Other solutions of the ϵ-conditions are possible. The one proposed above can be seen as the proposition corresponding to "$U(a)$ true & $a \in S$" which means "there exists b such that $b \in U(a)$ and $a \in S$". If we formalize it directly, we obtain $(\exists z \in U(a))\ \mathsf{Id}(S, a, a)$, which is exactly $U(a)\ \&\ \mathsf{Id}(S, a, a)$, by the definition of & (see [Mar84] p. 43). If we note that "there exists b such that $b \in U(a)$ and $a \in S$" is equivalent to "there exists $c \in \Sigma(S, U)$ such that $\mathsf{fst}(c) = a \in S$", we reach another solution for the ϵ-conditions, namely $(\exists z \in \Sigma(S, U))\ \mathsf{Id}(S, \mathsf{fst}(z), a)$ (see also section 5.3.4). However, the particular form of the solution is inessential, as long as it satisfies the ϵ-conditions. We thus put:

Definition 5.3.2 *Let S be any set and U any subset of S. If $(x : S)\ x\ \epsilon_S\ U$ is any propositional function satisfying the ϵ-conditions, we say that a is an element of U when $a\ \epsilon_S\ U$ is true.*

Since $a\ \epsilon_S\ U$ is a proposition for any $a \in S$ and $U \subseteq S$, the property of being an element of U respects equality of elements of S; in fact,

$$\text{(substitution of elements)} \qquad \frac{\mathsf{Id}(S, a, b)\ \mathsf{true} \quad a\ \epsilon_S\ U\ \mathsf{true}}{b\ \epsilon_S\ U\ \mathsf{true}}$$

is a consequence of the Id-elimination rule (see [NPS90], p. 64).

The few simple steps taken above are enough to develop a theory of subsets. The usual relations (like inclusion and extensional equality), operations on subsets (like finitary and infinitary union and intersection) and other usual tools (families indexed over a subset, quantifiers ranging over a subset, the image of a function between sets, functions defined on subsets, finite subsets, etc.) can be introduced in a straightforward way by means of the above ϵ-conditions and intuitionistic logic. We repeat such work here with some detail, of course not expecting to produce surprise, but to give a direct feeling (experience) that ϵ-conditions are really enough, and that they allow a complete formalization which is faithful to usual intuitions and practice.

In this way subset theory, even if type-less, is developed in a predicative way, a fact which is inherited directly from type theory.

5.3.3 Inclusion and equality between subsets

Given two subsets U and V of a set S, it is usual to say that U is included in V if every element of U is also an element of V. We thus put:

Definition 5.3.3 (Inclusion) *For any $U, V \subseteq S$, we define the inclusion of U into V by*

$$U \subseteq_S V \equiv (\forall x \in S)\ (x\ \epsilon_S\ U \to x\ \epsilon_S\ V)$$

Thus, contrary to $U \subseteq S$, $U \subseteq_S V$ is a proposition even if often, as in usual mathematical practice, we write $U \subseteq_S V$ to mean $U \subseteq_S V\ \mathsf{true}$.

By the first ϵ-condition, $U \subseteq_S V \leftrightarrow (\forall x \in S)(U(x) \to V(x))$ is true; this tells that $U \subseteq_S V$ could equivalently be defined as $(\forall x \in S)(U(x) \to V(x))$.

The usual basic rules connecting membership with inclusion are immediately derivable from the above definition by means of the ϵ-conditions; they confirm the understanding that $U \subseteq_S V$ is true if and only if every element of U is also an element of V.

Proposition 5.3.4 *For any S set and $U, V \subseteq S$, the following rules are derivable:*

$\subseteq_S$*-introduction*

$$\dfrac{\begin{array}{c}[x \;\epsilon_S\; U \;\mathsf{true}]_1 \\ | \\ x \;\epsilon_S\; V \;\mathsf{true}\end{array}}{U \subseteq_S V \;\mathsf{true}}\; 1$$

$\subseteq_S$*-elimination*

$$\dfrac{a \;\epsilon_S\; U \;\mathsf{true} \quad U \subseteq_S V \;\mathsf{true}}{a \;\epsilon_S\; V \;\mathsf{true}}$$

Proof. A derivation of $\subseteq_S$-introduction is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{S \;\mathsf{set} \quad U \subseteq S \quad [x \in S]_2}{x \;\epsilon_S\; U \;\mathsf{prop}}}{[x \;\epsilon_S\; U \;\mathsf{true}]_1}}{\begin{array}{c} | \\ x \;\epsilon_S\; V \;\mathsf{true}\end{array}}}{x \;\epsilon_S\; U \to x \;\epsilon_S\; V \;\mathsf{true}}\; 1}{U \subseteq_S V \;\mathsf{true}}\; 2$$

and a derivation of $\subseteq_S$-elimination is:

$$\dfrac{a \;\epsilon_S\; U \;\mathsf{true} \qquad \dfrac{U \subseteq_S V \;\mathsf{true} \qquad \dfrac{a \;\epsilon_S\; U \;\mathsf{true}}{a \in S}\;\text{second } \epsilon\text{-cond.}}{a \;\epsilon_S\; U \to a \;\epsilon_S\; V \;\mathsf{true}}\;\forall\text{-elim.}}{a \;\epsilon_S\; V \;\mathsf{true}}$$

Since $\subseteq_S$ is defined in terms of the connective of implication, it inherits all its properties. For instance, $\subseteq_S$ is a preorder on subsets, with a top and a bottom element:

Proposition 5.3.5 *For any S set and any $U, V, W \subseteq S$, the following hold:*

(reflexivity) $U \subseteq_S U$ *(transitivity)* $\dfrac{U \subseteq_S V \qquad V \subseteq_S W}{U \subseteq_S W}$

Moreover, putting $\top_S \equiv \{x \in S |\; \mathsf{Id}(S, x, x)\}$ and $\bot_S \equiv \emptyset_S \equiv \{x \in S |\; \neg\mathsf{Id}(S, x, x)\}$ we obtain

(top) $U \subseteq_S \top_S$ *(bottom)* $\emptyset_S \subseteq_S U$

While the first two statements are an immediate consequence of $\subseteq_S$-rules (and in turn of reflexivity and transitivity of implication), the second two follow by logic from

$$(\forall x \in S)\; (x \;\epsilon_S\; U \to \mathsf{Id}(S, x, x))\;\mathsf{true}$$

and by ex falso quodlibet, respectively, whatever propositional function U is.

Equality between subsets is usually defined by extensionality, that is, for any $U, V \subseteq S$, U and V are said to be equal if they have the same elements. We thus put:

Definition 5.3.6 (Extensional equality) *For any U, V subsets of the set S, we define extensional equality of U and V to be the proposition:*

$$U =_S V \equiv (\forall x \in S)\; (x \;\epsilon_S\; U \leftrightarrow x \;\epsilon_S\; V).$$

We say that the subset U is (extensionally) equal to the subset V if $U =_S V$ true.

The subsets U and V are (extensionally) equal if and only if for any $a \in S$, $a\ \epsilon_S\ U$ true iff $a\ \epsilon_S\ V$ true, and thus, by the first ϵ-condition, $U(a)$ true iff $V(a)$ true. Such equality must be distinguished from the stronger equality $U(x) = V(x)\ [x : S]$, which means that, for any $a \in S$, $b \in U(a)$ if and only if $b \in V(a)$, which is one of the basic judgments of type theory, and which could be called the *intensional* equality of the subsets U and V (since it requires U and V to have the same elements *and*, for each of them, with the same proofs).

By the definitions, it is immediate that the proposition

$$(U =_S V) \leftrightarrow (U \subseteq_S V\ \&\ V \subseteq_S U)$$

holds. Actually, $=_S$ is the equivalence relation on subsets induced by the preorder $\subseteq_S$ by forcing symmetry to hold. As for properties of $\subseteq_S$, the properties characterizing equivalences, in this case

reflexivity	$U =_S U$
symmetry	$U =_S V \rightarrow V =_S U$
transitivity	$U =_S V\ \&\ V =_S W \rightarrow U =_S W$

can also be seen as inherited from the properties of the logical connective $\leftrightarrow$.

Once the notion of equality has been clarified, the definition of the type of subsets of a given set S is completed:

Definition 5.3.7 (Power of a set) *For any set S, the type of all subsets of S equipped with extensional equality is called the power of S and is denoted by $\mathcal{P}S$.*

When a function (or operation) is to be defined on $\mathcal{P}S$, one must take care to check that it is well defined on $\mathcal{P}S$, that is, that it respects extensional equality; in the sequel this verification is sometime not spelled out.

5.3.4 Subsets as images of functions

The notion of subset can be further illustrated, after the introduction of extensional equality, by looking at it from a slightly different perspective.

For any set S, and any set I, a function $f(i) \in S\ [i : I]$ is usually associated with the subset of S whose elements are those $a \in S$ for which there exists $i \in I$ such that $\mathsf{Id}(S, f(i), a)$ true. Here this is achieved simply by defining the image of a function as follows:

Definition 5.3.8 (Image of a set along a function) *For any sets S and I, and for any function $f(i) \in S\ [i : I]$, the subset of S defined by:*

$$\mathsf{Im}_f[I] \equiv \{x \in S|\ (\exists i \in I)\ \mathsf{Id}(S, f(i), x)\}$$

is called the image of I along f. Other notations for $\mathsf{Im}_f[I]$ include $\{f(i)|\ i \in I\}$ and $f[I]$. More generally, given a function with n arguments $f(i_1, \ldots, i_n) \in S\ [i_1 : I_1, \ldots, i_n : I_n]$ the image of $I_1, \ldots, I_n$ along f is defined by

$$\mathsf{Im}_f[I_1, \ldots, I_n] \equiv \{x \in S|\ (\exists i_1 \in I_1) \ldots (\exists i_n \in I_n)\ \mathsf{Id}(S, f(i_1, \ldots, i_n), x)\}$$

The definition of image associates a subset of a set S with a function into S. Actually, this gives an alternative characterization of subsets since the converse can also be proved (see [Mar84], page 64). In fact, every subset U of S is extensionally equal to the image of some set I along some function $f(i) \in S\ [i : I]$ or, in more informal and suggestive words, we could say that subsets are just one function apart from sets:

Theorem 5.3.9 *Every subset U of a set S is extensionally equal to the image of the set $\Sigma(S, U)$ along the left projection $\mathsf{fst}(i) \in S\ [i : \Sigma(S, U)]$; in symbols,*

$$U =_S \mathsf{Im}_{\mathsf{fst}}[\Sigma(S, U)]$$

that is, by unwinding definitions,

$$U =_S \{x \in S|\ (\exists i \in \Sigma(S, U))\ \mathsf{Id}(S, \mathsf{fst}(i), x)\}$$

holds for every set S and $U \subseteq S$.

Proof. By the definitions and the ϵ-conditions, the claim $U =_S \mathsf{fst}[\Sigma(S,U)]$ becomes

$$(\forall x \in S)\ (U(x) \leftrightarrow (\exists y \in \Sigma(S,U))\ \mathsf{Id}(S, \mathsf{fst}(y), x))$$

To prove it, assume that a is an arbitrary element of S, and that $z \in U(a)$. Then $\langle a, z\rangle \in \Sigma(S,U)$, thus $\mathsf{fst}(\langle a, z\rangle) = a \in S$, hence $\mathsf{r}(a) \in \mathsf{Id}(S, \mathsf{fst}(\langle a, z\rangle), a)$, and therefore

$$\langle \langle a, z\rangle, \mathsf{r}(a)\rangle \in (\exists y \in \Sigma(S,U))\ \mathsf{Id}(S, \mathsf{fst}(y), a)$$

This proves that $\lambda z.\langle \langle a, z\rangle, \mathsf{r}(a)\rangle$ is the term making $U(a) \to (\exists y \in \Sigma(S,U))\ \mathsf{Id}(S, \mathsf{fst}(y), a)$ true.

To prove the converse, assume $z \in (\exists y \in \Sigma(S,U))\ \mathsf{Id}(S, \mathsf{fst}(y), a)$. Then $\mathsf{fst}(z) \in \Sigma(S,U)$ and hence $\mathsf{snd}(\mathsf{fst}(z)) \in U(\mathsf{fst}(\mathsf{fst}(z)))$ which, together with the fact that $\mathsf{snd}(z) \in \mathsf{Id}(S, \mathsf{fst}(\mathsf{fst}(z)), a)$, gives $\mathsf{subst}(\mathsf{snd}(\mathsf{fst}(z)), \mathsf{snd}(z)) \in U(a)$, as wished (see [NPS90], p. 64).

The theorem above gives further evidence to the fact that the notion of being an element of a subset is the result of disregarding some information. Given a function $f(i) : S\ [i : I]$, the subset $\mathsf{Im}_f[I]$ can be seen as the result of a process with two different abstraction steps. First, we realize that to know that a is an element in $\mathsf{Im}_f[I]$ we can abstract on the particular argument i such that $\mathsf{Id}(S, f(i), a)\ \mathsf{true}$ and prove only $c \in (\exists i : I)\ \mathsf{Id}(S, f(i), a)$ for some c. Note however that, due to the constructive meaning of existential quantification in type theory, a specific element $i \in I$ such that $\mathsf{Id}(S, f(i), a)\ \mathsf{true}$ can immediately be obtained from c. So, the second step, where we really forget some information, is to say that a is in $\mathsf{Im}_f(I)$ if and only if $(\exists i : I)\ \mathsf{Id}(S, f(i), a)\ \mathsf{true}$.

Now let us consider the case of the function $\mathsf{fst}(z) \in S\ [z : \Sigma(S,U)]$, for some subset $U \subseteq S$. Then the above considerations bring to the conclusion that a is in $\mathsf{Im}_{\mathsf{fst}}[\Sigma(S,U)]$ if and only if $(\exists z : \Sigma(S,U))\ \mathsf{Id}(S, \mathsf{fst}(z), a)\ \mathsf{true}$. By the theorem above, $a\ \epsilon_S\ U\ \mathsf{true}$ is equivalent to a is in $\mathsf{Im}_{\mathsf{fst}}[\Sigma(S,U)]$, and hence also to $(\exists z : \Sigma(S,U))\ \mathsf{Id}(S, \mathsf{fst}(z), a)\ \mathsf{true}$. It is then interesting to observe that to pass from a given verification of $(\exists z : \Sigma(S,U))\ \mathsf{Id}(S, \mathsf{fst}(z), a)$ to the judgment $(\exists z : \Sigma(S,U))\ \mathsf{Id}(S, \mathsf{fst}(z), a)\ \mathsf{true}$ means to forget the verification making $U(a)\ \mathsf{true}$ without forgetting a, since a appears explicitly in the proposition itself. To supply all the details we left out amounts to find a proof of $(\exists z : \Sigma(S,U))\ \mathsf{Id}(S, \mathsf{fst}(z), a) \leftrightarrow U(a)\ \&\ \mathsf{Id}(S, a, a)\ \mathsf{true}$..

It is interesting to note that, since $U(a)\ \&\ \mathsf{Id}(S, a, a)$ is the "canonical" solution of the ϵ-conditions, the above equivalence gives an alternative, and more formal, proof of the fact that also the proposition $(\exists z : \Sigma(S,U))\ \mathsf{Id}(S, \mathsf{fst}(z), a)$ is a solution of the ϵ-conditions, as we already stated in section 5.3.2.

5.3.5 Singletons and finite subsets

Every element a of a set S is equal to any element b of the same set S making the propositional function $(x : S)\ \mathsf{Id}(S, x, a)$ true at b; such triviality means that for any $a \in S$ we can intuitively form the singleton $\{a\}$ by putting

$$\{a\} \equiv \{x \in S |\ \mathsf{Id}(S, x, a)\}$$

And then the idea is that a finite subset is the union of a finite number of singletons; so if $a_0, \ldots, a_{n-1} \in S$, for some natural number n, we put

$$\begin{array}{rcl} \{a_0, \ldots, a_{n-1}\} & \equiv & \{a_0\} \cup \{a_1\} \cup \ldots \cup \{a_{n-1}\} \\ & \equiv & \{x \in S |\ \mathsf{Id}(S, x, a_0) \vee \ldots \vee \mathsf{Id}(S, x, a_{n-1})\} \end{array}$$

But what does it mean, more precisely, to give $a_0, \ldots, a_{n-1} \in S$? It means that a is a function from $\mathsf{N}(n)$, a set with n elements, into S, and $a_0, \ldots, a_{n-1}$ are its values.

It is easy to define a family of sets $\mathsf{N}(n)\ \mathsf{set}\ [n : \mathsf{N}]$ such that $\mathsf{N}(0)$ has no elements and, for $n > 0$, the elements of $\mathsf{N}(n)$ are $0_n, \ldots, (n-1)_n$. Then a singleton is the image of a function $a : \mathsf{N}(1) \to S$, and a finite subset of S with n elements is the image of a function $a : \mathsf{N}(n) \to S$. We thus put:

Definition 5.3.10 (Singletons and finite subsets) *For every set S, a subset U of S is said to be finite if U is extensionally equal to the image of some function $a \in \mathsf{N}(n) \to S$, for some $n \in \mathsf{N}$ and in particular it is called a singleton if $n = 1$; more formally U is finite if*

$$(\exists z \in \Sigma(\mathsf{N}, (n)\ \mathsf{N}(n) \to S))\ (U =_S \mathsf{Im}_{\mathsf{snd}(z)}[\mathsf{N}(\mathsf{fst}(z))])\ \mathsf{true}$$

In particular, the empty subset of S is also finite, being equal to the image of a function from $\mathsf{N}(0)$ into S.

Given the above definition, the assertion "U is finite" is just a proposition with parameter U. This allows for instance to express rigorously in type theory a statement of the form "there exists a finite subset U_0 of U such that $\ldots U_0 \ldots$" by

$$(\exists z \in \Sigma(\mathsf{N}, (n)\ \mathsf{N}(n) \to S))\ \mathsf{Im}_{\mathsf{snd}(z)}[\mathsf{N}(\mathsf{fst}(z))] \subseteq_S U\ \&\ \ldots \mathsf{Im}_{\mathsf{snd}(z)}[\mathsf{N}(\mathsf{fst}(z))] \ldots$$

(a typical example is the definition of Stone cover in [Sam87]).

Proposition 5.3.11 *For any set S, if U is a finite subset of S, then either U is empty or there exist a natural number $n > 0$ and $a_0, \ldots, a_{n-1} \in S$ such that $U =_S \{a_0, \ldots, a_{n-1}\}$.*

Proof. The proof is nothing but working out definitions, using properties of finite sets, and fixing notation. U finite means that

$$(\exists z \in \Sigma(\mathsf{N}, (n)\ \mathsf{N}(n) \to S))\ (U =_S \mathsf{Im}_{\mathsf{snd}(z)}[\mathsf{N}(\mathsf{fst}(z))])\ \mathsf{true}$$

If w is one of its verifications then $\mathsf{fst}(w) \in \Sigma(\mathsf{N}, (n)\ \mathsf{N}(n) \to S))$, and so $n \equiv \mathsf{fst}(\mathsf{fst}(w))$ is a natural number and $a \equiv \mathsf{snd}(\mathsf{fst}(w))$ is a function in $\mathsf{N}(n) \to S$. Then $U =_S \mathsf{Im}_a[\mathsf{N}(n)]$ holds. If n is zero we have finished since $\mathsf{Im}_a[\mathsf{N}(n)]$ is empty. Otherwise, by definition of image, $x\ \epsilon_S\ \mathsf{Im}_a[\mathsf{N}(n)]\ \mathsf{true}$ if and only if $(\exists i \in \mathsf{N}(n))\ \mathsf{Id}(S, a(i), x)\ \mathsf{true}$. Then, writing a_i for $a(i_n)$, by the rule of $\mathsf{N}(n)$-elimination we have

$$(x\ \epsilon_S\ \mathsf{Im}_a[\mathsf{N}(n)]) \leftrightarrow (\mathsf{Id}(S, x, a_0) \vee \mathsf{Id}(S, x, a_1) \vee \ldots \vee \mathsf{Id}(S, x, a_{n-1}))\ \mathsf{true}$$

as wished.

Set theoretic operations can be defined among finite subsets which give a finite subset as result. For instance, suppose that U and V are finite subsets determined by the elements c and d in $\Sigma(\mathsf{N}, (n)\ \mathsf{N}(n) \to S)$, i.e. $U =_S \mathsf{Im}_{\mathsf{snd}(c)}[\mathsf{N}(\mathsf{fst}(c))]$ and $V =_S \mathsf{Im}_{\mathsf{snd}(d)}[\mathsf{N}(\mathsf{fst}(d))]$. Then the union of U and V is the finite subset determined by

$$\langle \mathsf{fst}(c) + \mathsf{fst}(d), \lambda x.\ \mathit{if}\ x < \mathsf{fst}(c)\ \mathit{then}\ \mathsf{snd}(c)(x)\ \mathit{else}\ \mathsf{snd}(d)(x - \mathsf{fst}(c)) \rangle$$

On the other hand, intersection between the finite subsets U and V, determined by c and d, cannot be determined by an element in $\Sigma(\mathsf{N}, (n)\ \mathsf{N}(n) \to S)$ unless equality among elements of S is decidable. In fact, suppose that there exists a function g such that $g(c, d) \in \Sigma(\mathsf{N}, (n)\ \mathsf{N}(n) \to S)$ determines the finite subset which corresponds to the intersection of U and V. Then consider the case in which U and V are the singletons $\{a\}$ and $\{b\}$ for $a, b \in S$, i.e. U and V are determined by $\langle 1, \lambda x.\ a \rangle$ and $\langle 1, \lambda x.\ b \rangle$ in $\Sigma(\mathsf{N}, (n)\ \mathsf{N}(n) \to S)$ respectively. Then the subset determined by $g(\langle 1, \lambda x.\ a \rangle, \langle 1, \lambda x.\ b \rangle)$ is either a singleton or empty according to whether $\mathsf{Id}(S, a, b)$ is true or not. Hence $\mathsf{fst}(g(\langle 1, \lambda x.\ a \rangle, \langle 1, \lambda x.\ b \rangle)) \in \mathsf{N}$ is equal to 1 if and only if $\mathsf{Id}(S, a, b)\ \mathsf{true}$, which allows to decide on the equality of a and b since equality in N is decidable[6].

Many usual properties of singletons and finite subsets are obtained by intuitionistic logic from the above definitions. We give the following proposition as a sample:

Proposition 5.3.12 *For any S set, $U \subseteq S$ and $a \in S$,*

$$a\ \epsilon_S\ U\ \mathsf{true}\ \mathit{iff}\ \{a\} \subseteq_S U\ \mathsf{true}$$

Proof. Assume $a\ \epsilon_S\ U\ \mathsf{true}$ and let $x\ \epsilon_S\ \{a\}\ \mathsf{true}$; then $\mathsf{Id}(S, x, a)\ \mathsf{true}$, and hence by the rule of substitution on elements $x\ \epsilon_S\ U\ \mathsf{true}$, so that, by $\subseteq_S$-introduction $\{a\} \subseteq_S U\ \mathsf{true}$. Conversely if $\{a\} \subseteq_S U\ \mathsf{true}$ then, by $\subseteq_S$-elimination, $a\ \epsilon_S\ U\ \mathsf{true}$ because obviously $a\ \epsilon_S\ \{a\}\ \mathsf{true}$.

However, some other common properties require new definitions to be expressed. An example is for instance $U =_S \bigcup_{a \epsilon_S U} \{a\}$, where the notion of union indexed over a subset is necessary (see section 5.3.9).

[6] A solution to the problem of intersection exists, but it requires a more complex definition of finite subset, for which proposition 5.3.11 fails. The intuitive idea is that, given a finite set J and, for any $j \in J$, a finite set $I(j)$, a subset is finite if it is extensionally equal to the subset $\{x \in S |\ \bigvee_{j \in J}(\bigwedge_{i \in I(j)} x = a_{ji})\}$. More formally, the finite subsets are determined by the elements of the set $\Sigma(\mathsf{N}, (n)\ \Sigma(\mathsf{N}(n) \to \mathsf{N}, (k)\ \Pi(\mathsf{N}(n), (x)\ \mathsf{N}(k(x)) \to S)))$. It can be shown that this definition reduces to the one in the main text if the equality of S is decidable.

5.3.6 Finitary operations on subsets

One of the main reasons for the definition of subsets as propositional functions is that it allows to define operations on subsets with a subset as value. We begin with usual set-theoretic operations.

Definition 5.3.13 (Finitary operations on subsets) *For any $U, V \subseteq S$, we define*

$$\begin{array}{lrl} \textit{intersection}: & U \cap V & \equiv \{x \in S |\ U(x)\ \&\ V(x)\} \\ \textit{union}: & U \cup V & \equiv \{x \in S |\ U(x) \vee V(x)\} \\ \textit{implication}: & U \Rightarrow V & \equiv \{x \in S |\ U(x) \rightarrow V(x)\} \\ \textit{opposite}: & -U & \equiv \{x \in S |\ \neg U(x)\} \end{array}$$

Note the common pattern of the above definitions: an operation on subsets, i.e. propositional functions, is obtained by lifting (through abstraction) a connective acting on propositions. More formally, if $\bullet$ is a given connective, then the corresponding operation on subsets $\circ$ is defined by

$$\circ \equiv (S : set)(U : (x : S)\ \mathsf{prop})(V : (x : S)\ \mathsf{prop})(x : S)(U(x) \bullet V(x))$$

and hence $\circ : (S : set)(U : (x : S)\ \mathsf{prop})(V : (x : S)\ \mathsf{prop})(x : S)\ \mathsf{prop}$. This is the direct link between "subset-theoretic" operations and intuitionistic logical connectives. It is also clear that all of the above operations on subsets respect extensional equality, by the logical meta-theorem of replacement of equivalent propositions.

The following proposition tells that each of them can be characterized in terms of elements in the expected, traditional way:

Proposition 5.3.14 *For any $U, V \subseteq S$ and any $a \in S$, the following hold*

$$\begin{array}{rcl} a\ \epsilon_S\ U \cap V\ \mathsf{true} & \textit{iff} & a\ \epsilon_S\ U\ \&\ a\ \epsilon_S\ V\ \mathsf{true} \\ a\ \epsilon_S\ U \cup V\ \mathsf{true} & \textit{iff} & a\ \epsilon_S\ U \vee a\ \epsilon_S\ V\ \mathsf{true} \\ a\ \epsilon_S\ U \Rightarrow V\ \mathsf{true} & \textit{iff} & a\ \epsilon_S\ U \rightarrow a\ \epsilon_S\ V\ \mathsf{true} \\ a\ \epsilon_S\ -U\ \mathsf{true} & \textit{iff} & \neg(a\ \epsilon_S\ U)\ \mathsf{true} \end{array}$$

Proof. Under the assumption $a \in S$, the judgment $a\ \epsilon_S\ U \cap V\ \mathsf{true}$ is equivalent to the judgement $((x : S)\ U(x)\ \&\ V(x))(a)\ \mathsf{true}$, that is $U(a)\ \&\ V(a)\ \mathsf{true}$, which in turn is equivalent to $a\ \epsilon_S\ U\ \&\ a\ \epsilon_S\ V\ \mathsf{true}$ by the first ϵ-condition.

Exactly the same argument applies to all other operations.

Even if $a\ \epsilon_S\ U$ and $U(a)$ are logically equivalent under the assumption that $a \in S$, note that it is the use of the ϵ-notation which allows to make evident an intuitive content which otherwise would be completely hidden in the syntactic rule of reduction by which for instance $(U\&V)(a)$ and $U(a)\ \&\ V(a)$ are just equal expressions. This is one of the main reasons for introducing it.

As for inclusion and equality, the properties of operations on subsets are an immediate consequence of the properties of the corresponding logical connective used to define them.

The logical rules of &-elimination say that

$$U \cap V \subseteq_S U \text{ and } U \cap V \subseteq_S V$$

while by &-introduction it is immediate that

$$\frac{W \subseteq_S U \quad W \subseteq_S V}{W \subseteq_S U \cap V}$$

and thus $U \cap V$ is the infimum of U and V with respect to the partial order $\subseteq_S$.

Similarly, by the $\vee$-rules, we have

$$U \subseteq_S U \cup V \text{ and } V \subseteq_S U \cup V$$

and

$$\frac{U \subseteq_S W \quad V \subseteq_S W}{U \cup V \subseteq_S W}$$

which say that $U \cup V$ is the supremum of U and V.

If instead of rules we consider logical truths, then it is immediate that

$$\begin{array}{lrcl} \text{associativity} & (U \cap V) \cap W & =_S & U \cap (V \cap W) \\ \text{commutativity} & U \cap V & =_S & V \cap U \\ \text{idempotency} & U \cap U & =_S & U \end{array}$$

hold, and that the same properties hold for $\cup$.

The link between $\Rightarrow$ and $\subseteq_S$, is given by

$$(U \Rightarrow V) =_S \top_S \quad \text{iff} \quad U \subseteq_S V$$

that is $(\forall x \in S)\ ((x\ \epsilon_S\ U \rightarrow x\ \epsilon_S\ V) \leftrightarrow \top)$ iff $(\forall x \in S)\ (x\ \epsilon_S\ U \rightarrow x\ \epsilon_S\ V)$, which is obvious because $(A \rightarrow B) \leftrightarrow \top$ is logically equivalent to $A \rightarrow B$, for any propositions A and B.

In general, the usual informal argument to prove a certain property of set-theoretic operations is perfectly reflected into a rigorous proof through intuitionistic logic.

5.3.7 Families of subsets and infinitary operations

We now turn to infinitary operations on subsets. The order of conceptual priority, however, is to deal before with families of subsets. The traditional notion of family of subsets has a simple definition in the present approach:

Definition 5.3.15 (Set-indexed family of subsets) *A family of subsets of S indexed by a set I is a propositional function $U : (i : I)(x : S)$ prop with two arguments, one in I and one in S. Applying U to an element i of I we obtain a propositional function $U(i)$ on elements of S, i.e. $U(i) \subseteq S$. Following traditional notation, given any $i \in I$, we put*

$$U_i \equiv U(i)$$

Hence the usual notation $(U_i)_{i \in I}$ can be used for a set-indexed family of subsets.

Two families of subsets U, V, indexed by I, are said to be equal if for any index $i \in I$ it is $U_i =_S V_i$, that is we put

$$((U_i)_{i \in I} =_S (V_i)_{i \in I}) \equiv (\forall i \in I)\ (U_i =_S V_i)$$

In other terms, U and V are equal if they are extensionally equal as binary relations between I and S, i.e. $(\forall i \in I)(\forall x \in S)\ (U(i, x) \leftrightarrow V(i, x))$.

Infinitary operations are easily defined on set-indexed families of subsets. Just as propositional connectives were used to define unary and binary operations, now quantifiers are used to define infinitary operations.

Definition 5.3.16 (Infinitary operations) *Let $(U_i)_{i \in I}$ be a set-indexed family of subsets of S. Then we put:*

$$\textstyle\bigcup_{i \in I} U_i \equiv \{x \in S |\ (\exists i \in I)\ U(i, x)\} \equiv (x : S)(\exists i \in I)\ U(i, x)$$

$$\textstyle\bigcap_{i \in I} U_i \equiv \{x \in S |\ (\forall i \in I)\ U(i, x)\} \equiv (x : S)(\forall i \in I)\ U(i, x)$$

Clearly $\bigcup_{i \in I} U_i$ and $\bigcap_{i \in I} U_i$ are subsets of S. Moreover, they behave in the expected way with respect to elements:

Proposition 5.3.17 *For any set-indexed family $(U_i)_{i \in I}$ of subsets of S, and any $a \in S$:*

$$a\ \epsilon_S \textstyle\bigcup_{i \in I} U_i\ \mathsf{true} \quad \textit{iff} \quad (\exists i \in I)\ (a\ \epsilon_S\ U_i)\ \mathsf{true}$$

$$a\ \epsilon_S \textstyle\bigcap_{i \in I} U_i\ \mathsf{true} \quad \textit{iff} \quad (\forall i \in I)\ (a\ \epsilon_S\ U_i)\ \mathsf{true}$$

Proof. The proof is perfectly similar to the proof of proposition 5.3.14.

The standard properties of union are obtained, as expected, from logical properties of the existential quantifier. Given any set-indexed family of subsets $(U_i)_{i\in I}$, for any $j \in I$ the $\exists$-introduction rule gives $(\forall x \in S)\ (x\ \epsilon_S\ U_j \to (\exists i \in I)\ x\ \epsilon_S\ U_i)\ \mathsf{true}$, which says that

$$\text{for all } j \in I, \quad U_j \subseteq_S \bigcup_{i\in I} U_i \tag{5.1}$$

Note that, since $U_j \subseteq_S \bigcup_{i\in I} U_i$ is a proposition and not a judgment, we could, more formally , express the above as $(\forall j \in I)\ (U_j \subseteq_S \bigcup_{i\in I} U_i)$.

Similarly, for any $x \in S$ and $W \subseteq S$, the rule of $\exists$-elimination

$$\frac{(\exists i \in I)\ x\ \epsilon_S\ U_i\ \mathsf{true} \qquad \begin{array}{c}[i \in I, x\ \epsilon_S\ U_i\ \mathsf{true}] \\ \vdots \\ x\ \epsilon_S\ W\ \mathsf{true}\end{array}}{x\ \epsilon_S\ W\ \mathsf{true}}$$

can be put in the form

$$\frac{(\forall i \in I)\ (x\ \epsilon_S\ U_i \to x\ \epsilon_S\ W)}{((\exists i \in I)\ x\ \epsilon_S\ U_i) \to x\ \epsilon_S\ W}$$

which says that

$$\frac{U_i \subseteq W \text{ for all } i \in I}{\bigcup_{i\in I} U_i \subseteq W} \tag{5.2}$$

Of course, the above two properties (5.1) and (5.2) say that union is the supremum of set-indexed families w.r.t. the order $\subseteq_S$.

An equivalent formulation of (5.1) and (5.2) together is

$$\bigcup_{i\in I} U_i \subseteq W \text{ iff for all } i \in I,\ U_i \subseteq W$$

which corresponds to

$$(\forall x \in S)\ ((\exists i \in I)\ x\ \epsilon_S\ U_i) \to x\ \epsilon_S\ W \text{ iff } (\forall i \in I)(\forall x \in S)\ (x\ \epsilon_S\ U_i \to x\ \epsilon_S\ W)$$

which is true by the intuitionistic laws of permutation of quantifiers with implication. One can actually prove a somewhat stronger statement, namely

$$(\forall x \in S)\ (((\exists i \in I)\ (x\ \epsilon_S\ U_i) \to x\ \epsilon_S\ W) \leftrightarrow (\forall i \in I)\ (x\ \epsilon_S\ U_i \to x\ \epsilon_S\ W))$$

which can also be expressed in terms of subsets, as

$$(\bigcup_{i\in I} U_i \Rightarrow W) =_S \bigcap_{i\in I} (U_i \Rightarrow W)$$

and shows the use of the subset operation $\Rightarrow$.

Quite similarly, from the rules for $\forall$, one obtains that intersection is the infimum of a set-indexed family $(U_i)_{i\in I}$.

5.3.8 The power of a set

In this section some facts specific of the type of subsets of a set S, equipped with extensional equality, will be illustrated. Let us stress that the type we are considering is *not* the type of the propositional functions over S, even if a subset of S is the same as a propositional function over S. In fact, a type is determined both by its elements and its equality relation, and we do not consider intensional equality between propositional functions as in [Mar84], but extensional equality as defined in definition 5.3.6.

First of all, we want to analyze the structure of $\mathcal{P}S$, equipped with finitary and infinitary operations, in algebraic terms. The fact that $\mathcal{P}S$ is equipped with extensional equality gives as a consequence that inclusion $\subseteq_S$ is a partial order on $\mathcal{P}S$.

Moreover, $(\mathcal{P}S, \cap)$ and $(\mathcal{P}S, \cup)$ are semi-lattices[7] because of the results in section 5.3.6. To show that $(\mathcal{P}S, \cap, \cup)$ is a lattice, we have to check that $\subseteq_S$ is the partial order induced by the semi-lattice operations $\cap$ and $\cup$, i.e.

$$U \cap V =_S U \text{ iff } U \subseteq_S V \text{ iff } U \cup V =_S V$$

The first equivalence is immediate by logic (and proposition 5.3.14) once we expand definitions into $(\forall x \in S)\ (x\ \epsilon_S\ U \cap V \leftrightarrow x\ \epsilon_S\ U)$ if and only if $(\forall x \in S)\ (x\ \epsilon_S\ U \rightarrow x\ \epsilon_S\ V)$. Similarly, the second equivalence holds because $A \vee B \leftrightarrow B$ iff $A \rightarrow B$, for any propositions A and B.

The next step is to show that $\mathcal{P}S$ is a complete lattice with respect to infinitary union and intersection. The traditional definition is that a lattice $\mathbf{L}$ is complete if any family $(f_i)_{i\in I}$, where I is a set, of elements of $\mathbf{L}$ has a supremum. To express this inside type theory, we lack only the definition of set-indexed family of elements in a type (or in a set):

Definition 5.3.18 (set-indexed family of elements) *Let C be any type or set. A set-indexed family of elements of C is a function f defined on a set I with values in C. As usual, the notation, $(f_i)_{i\in I}$, where $f_i \equiv f(i)$, is used.*

We already used set-indexed family of elements of a type within this chapter in section 5.3.7, where we introduced the notion of set-indexed family of subsets of a set. In general, the foundational reason for introducing set-indexed families of elements of a type is that they allow to give a meaning to quantification over the elements of some sub-types. In fact, given a function f from the set I into the type C, the quantification over the image of f is reduced to a quantification over the set of indexes I. An example coming from mathematical practice is in [SVV96], where we introduced set-based Scott domains, i.e. Scott domains such that the type of compact elements can be indexed by a set.

Now, the definition of complete lattice in our approach is literally as above, but one must be careful that it has a different meaning according to the foundational attitude. In the classical view, any sub-type of $\mathcal{P}S$ can be indexed by a set, while we expect this to be false in type theory. We believe, however, that from a computational point of view it is necessary, but in the same time sufficient, to consider only families of subsets which are set-indexed.

Hence $\mathcal{P}S$ is a complete lattice because we have shown in section 5.3.7 that any set-indexed family of subsets has both supremum and infimum. It is now easy to prove also:

Theorem 5.3.19 *For any set S, $\mathcal{P}S = \langle \mathcal{P}S, \cap, \bigcup, \top_S, \bot_S \rangle$ is a frame (alias locale, complete Heyting algebra).*

Proof. After the preceding results, it remains to be proved only that infinitary union distributes over intersection, that is:

$$(\bigcup_{i\in I} U_i \cap W) =_S \bigcup_{i\in I} (U_i \cap W)$$

It is immediate to see that this correspond exactly to a logical law of quantifier shifting, namely

$$(\forall x \in S)\ ((\exists i \in I)\ x\ \epsilon_S\ U_i\ \&\ x\ \epsilon_S\ W \leftrightarrow (\exists i \in I)\ (x\ \epsilon_S\ U_i\ \&\ x\ \epsilon_S\ W))$$

As an example of how a classical theorem is rendered in our notion of power of a set, we give here a constructive version of Cantor's diagonalization theorem:

Theorem 5.3.20 (Cantor's diagonalization) *Let S be any set. Then for any set-indexed family $(F_x)_{x\in S}$ of subsets of S, there is a subset $D_F \subseteq S$ which is extensionally different from F_x for any $x \in S$.*

[7] Here and in the whole chapter we adhere to the principle of adopting standard algebraic terminology for structures $(A, f_1, \ldots, f_n)$, where A is a type, and not necessarily a set.

Proof. Given the family $(F_x)_{x\in S}$, i.e. $F(x,y)$ prop $[x:S, y:S]$, put $D_F \equiv (y:S)\ \neg F(y,y)$, that is, $D_F(y) \equiv \neg F(y,y)$. For any $x \in S$, $D_F =_S F_x$ would mean that $(\forall y \in S)\ \neg F(y,y) \leftrightarrow F(x,y)$, which for $y = x$ would give $\neg F(x,x) \leftrightarrow F(x,x)$, which is a contradiction. So for any $x \in S$, it is $\neg(D_F =_S F_x)$

Another example, inspired by topos theory, is the bi-univocal correspondence between $\mathcal{P}S$ and $S \to \mathcal{P}\mathsf{N}_1$, the collection of families of subsets of the one-element set N_1 equipped with equality as defined in 5.3.15. We leave the details.

5.3.9 Quantifiers relative to a subset

The meaning of quantification over a subset U of a set S is that the range of quantification is restricted to elements of U, rather than all elements of S. A common definition in pure logic is that of quantifiers relative to a property; the idea is to adapt it to type theory in such a way to make it visible that U is considered as the domain of quantification.

Definition 5.3.21 (Quantifiers relative to a subset) *Let S be a set and $U \subseteq S$. Then, for any propositional function $A(x)$ prop $[x:S, x\ \epsilon_S\ U$ true$]$ we put:*

$$(\forall x\ \epsilon_S\ U)\ A(x) \equiv (\forall x \in S)\ (x\ \epsilon_S\ U \to A(x))$$
$$(\exists x\ \epsilon_S\ U)\ A(x) \equiv (\exists x \in S)\ (x\ \epsilon_S\ U\ \&\ A(x))$$

The operators $(\forall x\ \epsilon_S\ U)$ and $(\exists x_S\ \epsilon_S\ U)$ are called, respectively, the universal and existential quantifier relative to U.

Note that the above definition makes use of the fact, specific to type theory, that $A \to B$ and $A\ \&\ B$ are propositions provided that A is a proposition and B is a proposition under the assumption that A is true.

It is an easy matter now to check that quantifiers relative to a subset U obey to rules completely similar to those for quantifiers in intuitionistic logic, but with explicit mention of the domain of quantification, as in [Mar84]:

$\forall$-introduction

$$\frac{\begin{array}{c}[x\ \epsilon_S\ U\ \mathsf{true}]\\ \vdots\\ A(x)\ \mathsf{true}\end{array}}{(\forall x\ \epsilon_S\ U)\ A(x)\ \mathsf{true}}$$

$\forall$-elimination

$$\frac{a\ \epsilon_S\ U\ \mathsf{true} \quad (\forall x\ \epsilon_S\ U)\ A(x)\ \mathsf{true}}{A(a)\ \mathsf{true}}$$

$\exists$-introduction

$$\frac{a\ \epsilon_S\ U\ \mathsf{true} \quad A(a)\ \mathsf{true}}{(\exists x\ \epsilon_S\ U)\ A(x)\ \mathsf{true}}$$

$\exists$-elimination

$$\frac{(\exists x\ \epsilon_S\ U)\ A(x)\ \mathsf{true} \qquad \begin{array}{c}[x\ \epsilon_S\ U\ \mathsf{true}, A(x)\ \mathsf{true}]\\ \vdots\\ C\ \mathsf{true}\end{array}}{C\ \mathsf{true}}$$

Such rules are only abbreviations for deductions in type theory. For instance, the $\forall$-introduction rule relativized to U is an abbreviation of

$$\cfrac{\cfrac{\cfrac{\cfrac{\begin{array}{c}S\ \mathsf{set} \quad U \subseteq S \quad [x \in S]_2\\ \searrow\ |\ \swarrow\\ x\ \epsilon_S\ U\ \mathsf{prop}\end{array}}{[x\ \epsilon_S\ U\ \mathsf{true}]_1}}{\begin{array}{c}\vdots\\ A(x)\ \mathsf{true}\end{array}}}{x\ \epsilon_S\ U \to A(x)\ \mathsf{true}}\ 1}{(\forall x\ \epsilon_S\ U)A(x)\ \mathsf{true}}\ 2$$

Once we have access to quantifiers relative to subsets, many of the notions defined on sets can be extended to subsets in a straightforward way; we now see the case of arbitrary unions and intersections. Before that, the notion of set-indexed family of subsets must be generalized to subset-indexed families.

Definition 5.3.22 (Subset-indexed family of subsets) *Let S and I be two sets and U be a subset of I. Then a propositional function*

$$V : (i : I)(y : U(i))(x : S) \ \mathsf{prop}$$

is said to be a family of subsets of S indexed on the subset U if the truth of $V(i, y, x)$ does not depend on y, i.e. $V(i, y_1) =_S V(i, y_2)$ for any $y_1, y_2 \in U(i)$; then one can hide the variable y and write

$$V_i \subseteq S \ [i \ \epsilon_I \ U \ \mathsf{true}].$$

The infinitary operations of union and intersection are immediately extended to subset-indexed families of subsets, simply by replacing quantifiers with quantifiers relative to a subset. So, if $V_i \subseteq S \ [i \ \epsilon_I \ U \ \mathsf{true}]$, we put

$$\bigcup_{i \epsilon_I U} V_i \equiv \{x : S |\ (\exists i \ \epsilon_I \ U) \ x \ \epsilon_S \ V_i\} \equiv (x : S)(\exists i \in I) \ (i \ \epsilon_I \ U \ \& \ x \ \epsilon_S \ V_i)$$

and

$$\bigcap_{i \epsilon_I U} V_i \equiv \{x : S |\ (\forall i \ \epsilon_I \ U) \ x \ \epsilon_S \ V_i\} \equiv (x : S)(\forall i \in I) \ (i \ \epsilon_I \ U \rightarrow x \ \epsilon_S \ V_i)$$

As an exercise, we can prove here the property we left out in section 5.3.5.

Proposition 5.3.23 *For any S set and $U \subseteq S$,*

$$U =_S \bigcup_{i \epsilon_S U} \{i\}$$

Proof. The subset-indexed family is of course $\{i\} \subseteq S \ [i \ \epsilon_S \ U]$. In fact, for any $x \in S$, we have $x \ \epsilon_S \ \bigcup_{i \epsilon_S U} \{i\}$ true if and only if $(\exists i \ \epsilon_S \ U) \ x \ \epsilon_S \ \{i\}$ true if and only if $(\exists i \ \epsilon_S \ U)$ $\mathsf{Id}(S, x, i)$ true if and only if $(\exists i \in S) \ i \ \epsilon_S \ U \ \& \ \mathsf{Id}(S, x, i)$ true if and only if $x \ \epsilon_S \ U$ true.

We propose a second exercise: prove that if $U \subseteq_I W$ and $V_i \subseteq S \ [i : I]$, then

$$\bigcup_{i \epsilon_I U} V_i \subseteq_S \bigcup_{i \epsilon_I W} V_i$$

A similar result holds also in the weaker assumption $V_i \subseteq S \ [i \ \epsilon_I \ W]$, but with a more complicated statement and proof.

5.3.10 Image of a subset and functions defined on a subset

The idea of relativized quantifiers, makes it natural to extend to subsets also the notion of image of a set:

Definition 5.3.24 (Image of a subset) *Let S and I be sets. Then, given any function*

$$f(i) \in S \ [i : I]$$

and any subset U of I, the subset of S defined by:

$$f[U] \equiv \{x \in S |\ (\exists i \ \epsilon_I \ U) \ \mathsf{Id}(S, f(i), x)\} \equiv (x : S)(\exists i \in I) \ (i \ \epsilon_I \ U \ \& \ \mathsf{Id}(S, f(i), x))$$

is called the image of U along f. *An alternative notation for the image of U along f is $\{f(x) |\ U(x)\}$.*

More generally, given a function $f(x_1, \ldots, x_n) \in S \ [x_1 : I_1, \ldots, x_n : I_n]$ and a relation $R(x_1, \ldots, x_n)$ prop $[x_1 : I_1, \ldots, x_n : I_n]$ both with n arguments, the image of R along f is defined by

$$\mathsf{Im}_f[R] \equiv (x : S)(\exists i_1 \in I_1) \ldots (\exists i_n \in I_n) \ (R(i_1, \ldots, i_n) \ \& \ \mathsf{Id}(S, f(i_1, \ldots, i_n), x))$$

Alternative notations for $\mathsf{Im}_f[R]$ are $f[R]$ and $\{f(x_1, \ldots, x_n) |\ R(x_1, \ldots, x_n)\}$.

Of course, if U is the trivial subset $\top_I$, then $f[\top_i] =_S \mathsf{Im}_f[I]$. In general, all the expected properties can easily be checked. For instance, for any $U, V \subseteq I$,

$$\frac{U \subseteq_I V}{f[U] \subseteq_S f[V]}$$

follows immediately from definitions by intuitionistic logic. Another instructive exercise is to realize that $f[U] \equiv \cup_{i \epsilon_I U}\{f(i)\}$.

It is also worthwhile to notice that the image $f[U]$ is always extensionally equal to the image $\mathsf{Im}_g[J]$ of some set J along some function g: it is enough to consider $J \equiv \Sigma(I, U)$ and $g \equiv \lambda x.\ f(\mathsf{fst}(x))$.

If n subsets $U_1 \subseteq I_1, \ldots, U_n \subseteq I_n$ are given, then the image of $U_1, \ldots, U_n$ under f is obtained as a special case, by putting

$$R(i_1, \ldots, i_n) \equiv i_1 \ \epsilon_{I_1}\ U_1 \ \& \ldots \& \ i_n \ \epsilon_{I_n}\ U_n$$

For instance, given an operation $\cdot : S^2 \to S$, and writing as usual $b \cdot c$ for $\cdot(b, c)$, the image of the two subsets $U, V \subseteq S$ is the subset

$$(x : S)(\exists b, c \in S)\ (b \ \epsilon_S\ U \ \&\ c \ \epsilon_S\ V \ \&\ \mathsf{Id}(S, x, b \cdot c))$$

that is

$$(x : S)(\exists b \ \epsilon_S\ U)(\exists c \ \epsilon_S\ V)\ \mathsf{Id}(S, x, b \cdot c)$$

which, following the above conventions, is written also $\{b \cdot c |\ b \ \epsilon_S\ U, c \ \epsilon_S\ V\}$ or $\cdot[U, V]$; it is the latter notation which gives raise to $U \cdot V$, which is the standard notation for such subset used in algebra to mean, for instance, the product of ideals $I \cdot J$ or of subgroups $H \cdot K$, and which we found useful in formal topology.

The notion of function itself can be relativized to a subset in the following sense:

Definition 5.3.25 (Function defined on a subset) *If S is a set, I is a set and $U \subseteq I$, a function of two arguments $f(i, y) \in S\ [i : I, y : U(i)]$ is said to be a function from U to S, if the value $f(i, y)$ does not depend on y, that is if $(\forall y, y' \in U(i))\mathsf{Id}(S, f(i, y), f(i, y'))\ \mathsf{true}$; then one can hide the variable y and write simply*

$$f(i) \in S\ [i \ \epsilon_I\ U\ \mathsf{true}].$$

The intuitive content of such definition is that, just like the notion of element of a subset U is obtained by "forgetting" the witness y of $U(i)$, so a function f relativized to U is obtained by "forgetting" the second argument of the input. This of course can be done only when the specific value of y is irrelevant for the computation of $f(i, y)$, i.e. when $f(i, y)$ and $f(i, y')$ have the same value for any $y, y' \in U(i)$ as required above.

A similar definition can be given also when f is a function from I and $U(i)\ \mathsf{set}\ [i : I]$ into a type C. In this case, to express the fact that f does not depend on the second argument, the equality in C must be used instead of propositional equality, and thus, in general, the condition can not be expressed by a proposition.

Extending the previous terminology to functions defined on subsets, a function $f(i) \in C\ [i \ \epsilon_I\ U]$ is also called a *subset-indexed family* of elements of C. The remark following definition 5.3.18 applies here equally well. Again, examples are to be found in [SVV96].

Chapter 6

Development of non-trivial programs

6.1 Summary

In this chapter we will show how a non-trivial program schema can be formally developed within Martin-Löf's type theory. With respect to the other chapters we will use here also some types that are not described in appendix B but can be found in [NPS90].

6.2 Introduction

Since the 70s Martin-Löf has developed, in a number of successive variants, an Intuitionistic Theory of Types [Mar84, NPS90] (ITT for short in the following). The initial aim was to provide a formal system for constructive mathematics but the relevance of the theory also in computer science was soon recognized. In fact, from an intuitionistic perspective, defining a constructive set theory is equivalent to defining a logical calculus [How80] or a language for problem specification [Kol32]. Hence the topic is of immediate relevance both to mathematicians, logicians and computer scientists. Moreover, since an element of a set can also be seen as a proof of the corresponding proposition or as a program which solves the corresponding problem, ITT is also a functional programming language with a very rich type structure and an integrated system to derive correct programs from their specification [PS86]. These pleasant properties of the theory have certainly contributed to the interest for it arisen in the computer science community, especially among those people who believe that program correctness is a major concern in the programming activity [BCMS89]. Many are the peculiarities of the theory which justify this wide concern. As regards computing science, through very powerful type-definition facilities and the embedded principle of "propositions as types" it primarily supplies means to support the development of proved-correct programs. Indeed here type checking achieves its very aim, namely that of avoiding *logical* errors. There are a lot of works (see for instance [Nor81, PS86]) stressing how it is possible to write down, within the framework of type theory, the formal specification of a problem and then develop a program meeting this specification. Actually, examples often refer to a single, well-known algorithm which is formally derived within the theory. The analogy between a mathematical constructive proof and the process of deriving a correct program is emphasized. Formality is necessary, but it is well recognized that the master-key to overcome the difficulties of formal reasoning, in mathematics as well as in computer science, is abstraction and generality. Abstraction mechanisms are very well offered by type theory by means of assumptions and dependent types. In this chapter we want to emphasize this characteristic of the theory.

Instead of specifying a single problem we specify classes of problems and develop general solutions for them. Correctness of any specific instance is clearly assured by the theory. Another aspect we think of some interest in the chapter is the introduction of some new dependent types; especially the type $\mathsf{Tree}(A)$ of finite trees labeled by elements of A. This type has been defined by means of the W-type: the type of well-orderings.

Let us simply recall that if B is a type and $C(x)$ is a family of types, where x ranges on B, then $\mathsf{W}(B, C)$ can be thought of as the set of all well-founded trees with labels in B and branches determined by $C(x)$. More precisely if a node is labeled by $b \in B$ then the branches coming out from that node correspond to the elements of $C(b)$. Transfinite induction gives us the elimination rule for elements of this type: if, from the assumption that a proposition holds for the predecessors (sub-trees) of an element $t \in \mathsf{W}(B, C)$ it is possible to derive that it holds for t, then it holds for any element in $\mathsf{W}(B, C)$.

The chapter is organized as follows. In section 6.3, we begin the description of the types which allow to express the following game problems within type theory. First the type $\mathsf{Seq}(A)$, of finite sequences of elements in A, the type $\mathsf{Tree}(A)$ and some general functions defined on them are presented. Then $\mathsf{Graph}(A)$, the type of finitary directed graphs is introduced and it is shown how its elements can be viewed as laws for building up finite trees. Theorems characterizing this class of trees follow. In section 6.4 the problems we deal with are defined and their solutions developed. They are problems on "games" and clearly their specification strongly depends on how we describe a game. Finally two simple examples, based on the knight's tour problem are considered and discussed. Finally, in section 6.7 there are most of the details on the constructions we use.

6.3 Basic Definitions

In this section we will introduce some general types we will use in the next of the chapter. Let us recall that in section 6.7.1 you will find the definition of the type $\mathsf{N}_<(a)$, that is, given $a \in \mathsf{N}$, the type of all the natural number less than a, that is $0, \ldots, a-1$, and such that $\mathsf{N}_<(0) = \emptyset$ holds.

6.3.1 The set $\mathsf{Seq}(A)$

Instead of introducing the type of the lists on A directly, we will implement them as pairs since this approach makes the following easier. The first component of a pair is the length $n \in \mathsf{N}$ of the list and the second a function mapping an element $i \in \mathsf{N}_<(n)$ to the i-th element of the list. Thus, we put

$$\mathsf{Seq}(A) \equiv (\exists x \in \mathsf{N})\ \mathsf{N}_<(x) \to A$$

and, supposing $a \in A$ and $s \in \mathsf{Seq}(A)$, we make the following definitions:

$$\begin{array}{rcl} \mathsf{nil} & \equiv & \langle 0, \lambda x.\ \mathsf{R}_0(x)\rangle \\ a \bullet s & \equiv & \langle \mathsf{s}(\mathsf{fst}(s)), \lambda x.\ \text{if } x <^{\mathsf{N}} \mathsf{fst}(s) \text{ then } \mathsf{snd}(s)[x] \text{ else } a\rangle \end{array}$$

which implies that, if

$$\begin{array}{l} C(x)\ \mathsf{prop}\ [x : \mathsf{Seq}(A)] \\ d \in C(\mathsf{nil}) \\ e(x, y, z) \in C(x \bullet y)\ [x : A, y : \mathsf{Seq}(A), z : C(y)], \end{array}$$

then

$$\mathsf{List}_{rec}(s, d, e) \equiv \mathsf{E}(s, (n, f)\ \mathsf{N}_{rec}(n, d, (u, v)\ e(f[u], \langle u, f\rangle, v)))$$

is a correct proof-method to find a proof of $C(s)$.

We will use the abbreviations

$$\begin{array}{rcll} \sharp s & \equiv & \mathsf{fst}(s) & \text{for the length of } s \text{ and} \\ s\{i\} & \equiv & \mathsf{snd}(s)[i] & \text{for the } i\text{-th element of } s \end{array}$$

If $a \in A$ and $s \in \mathsf{Seq}(A)$ then the proposition $a\ \mathsf{InSeq}\ s$ holds if and only if a is an element of the sequence s. The following equations hold:

$$\left\{ \begin{array}{rcl} a\ \mathsf{InSeq}\ \mathsf{nil} & = & \bot \\ a\ \mathsf{InSeq}\ b \bullet t & = & (a =^A b) \vee (a\ \mathsf{InSeq}\ t) \end{array} \right.$$

where $a =^A b$ is a shorthand for $\mathsf{Id}(A, a, b)$ (see sppendix B). The solution is

$$a\ \mathsf{InSeq}\ s \equiv \mathsf{List}_{rec}(s, \bot, (x, y, z)\ (a =^A x)\ \vee\ z)$$

To filter out, from a given sequence s, all the elements which do not satisfy a given condition $f(x) \in \mathsf{Boole}\ [x : A]$ we define the function

$$\mathsf{filter}(f, s) \in \mathsf{Seq}(A)\ [f : (x : A)\ \mathsf{Boole}, s : \mathsf{Seq}(A)]$$

such that, for all $a \in A$, $a\ \mathsf{InSeq}\ \mathsf{filter}(f, s)$ if and only if both $f(a) =^{\mathsf{Boole}} \mathsf{true}$ and $a\ \mathsf{InSeq}\ s$. The recursive equations for the function filter are

$$\begin{cases} \mathsf{filter}(f, \mathsf{nil}) & = & \mathsf{nil} \\ \mathsf{filter}(f, a \bullet s) & = & \text{if } f(a) \text{ then } a \bullet \mathsf{filter}(f, s) \text{ else } \mathsf{filter}(f, s) \end{cases}$$

and they can be solved by making the explicit definition

$$\mathsf{filter}(f, s) \equiv \mathsf{List}_{rec}(s, \mathsf{nil}, (x, y, z) \text{ if } f(x) \text{ then } x \bullet z \text{ else } z)$$

Theorem 6.3.1 *If*

$$\begin{array}{l} a \in A \\ s \in \mathsf{Seq}(A) \\ f(x) \in \mathsf{Boole}\ [x : A] \end{array}$$

then

$$a\ \mathsf{InSeq}\ \mathsf{filter}(f, s)\ \textit{is true if and only if}\ (f(a) =^{\mathsf{Boole}} \mathsf{true})\ \&\ (a\ \mathsf{InSeq}\ s)$$

Proof. The proof is by list-induction over s. In the base case

$$a\ \mathsf{InSeq}\ \mathsf{filter}(f, \mathsf{nil}) \Longleftrightarrow (f(a) =^{\mathsf{Boole}} \mathsf{true})\ \&\ (a\ \mathsf{InSeq}\ \mathsf{nil})$$

must be proven. But this is obvious, since

$$\mathsf{filter}(f, \mathsf{nil}) = \mathsf{nil} \in \mathsf{Seq}(A) \text{ and } a\ \mathsf{InSeq}\ \mathsf{nil} = \bot.$$

In the induction step, assume that

$$a\ \mathsf{InSeq}\ \mathsf{filter}(f, s) \Longleftrightarrow (f(a) =^{\mathsf{Boole}} \mathsf{true})\ \&\ (a\ \mathsf{InSeq}\ s);$$

the goal is to prove

$$a\ \mathsf{InSeq}\ \mathsf{filter}(f, b \bullet s) \Longleftrightarrow (f(a) =^{\mathsf{Boole}} \mathsf{true})\ \&\ (a\ \mathsf{InSeq}\ b \bullet s).$$

The proof will proceed by a Boole-elimination on $f(b)$. Suppose $f(b) =^{\mathsf{Boole}} \mathsf{true}$, then we must prove

$$a\ \mathsf{InSeq}\ b \bullet \mathsf{filter}(f, s) \Longleftrightarrow (f(a) =^{\mathsf{Boole}} \mathsf{true})\ \&\ ((a =^{A} b)\ \vee\ (a\ \mathsf{InSeq}\ s)),$$

that is:

$$(a =^{A} b)\ \vee\ (a\ \mathsf{InSeq}\ \mathsf{filter}(f, s)) \Longleftrightarrow (a =^{A} b) \vee ((f(a) =^{\mathsf{Boole}} \mathsf{true})\ \&\ (a\ \mathsf{InSeq}\ s))$$

that obviously holds by induction hypothesis.

In the second case, if $f(b) =^{\mathsf{Boole}} \mathsf{false}$, we must prove

$$a\ \mathsf{InSeq}\ \mathsf{filter}(f, s) \Longleftrightarrow (f(a) =^{\mathsf{Boole}} \mathsf{true})\ \&\ ((a =^{A} b)\ \vee\ (a\ \mathsf{InSeq}\ s))$$

that is, by inductive hypothesis, our goal is to prove

$$\begin{array}{l} (f(a) =^{\mathsf{Boole}} \mathsf{true})\ \&\ (a\ \mathsf{InSeq}\ s) \Longleftrightarrow \\ \qquad\qquad (f(a) =^{\mathsf{Boole}} \mathsf{true})\ \&\ ((a =^{A} b)\ \vee\ (a\ \mathsf{InSeq}\ s)). \end{array}$$

One implication is obvious. To obtain the proof of the other one, assume

$$(f(a) =^{\mathsf{Boole}} \mathsf{true})\ \&\ ((a =^{A} b)\ \vee\ (a\ \mathsf{InSeq}\ s)),$$

then we have

$$(a =^{A} b)\ \vee\ (a\ \mathsf{InSeq}\ s)$$

by &-elimination. Then the result follows by $\vee$-elimination since $(a =^{A} b)$ yields $f(a) =^{\mathsf{Boole}} f(b)$, by extensionality, and hence $\mathsf{true} = \mathsf{false} \in \mathsf{Boole}$; but then one obtains $(f(a) =^{\mathsf{Boole}} \mathsf{true})\ \&\ (a\ \mathsf{InSeq}\ s)$, by $\bot$-elimination; on the other hand, if $(a\ \mathsf{InSeq}\ s)$ then $(f(a) =^{\mathsf{Boole}} \mathsf{true})\ \&\ (a\ \mathsf{InSeq}\ s)$, follows by &-introduction.

6.3.2 The set $\mathsf{Tree}(A)$

The set $\mathsf{Tree}(A)$ is the set of all finite trees whose nodes are labeled with elements in the set A. As we saw earlier, the well-ordering type has labeled trees of finite depth as elements, so to obtain finite trees we only have to add the constrain that any node has a finite set of predecessors. We make the following definition

$$\mathsf{Tree}(A) \equiv \mathsf{W}(A \times \mathsf{N}, (x)\ \mathsf{N}_<(\mathsf{snd}(x)))$$

which is the set of finitely branched trees with nodes of the form $\langle a, n\rangle$ where a is an element in A and n is the number of immediate successors.

Then, for instance, the singleton tree with only one node labeled by $a_0 \in A$, can be defined as

$$\mathsf{leaf}(a_0) \equiv \mathsf{sup}(\langle a_0, 0\rangle, \mathsf{empty}) \qquad \bullet\ a_0$$

where $\mathsf{empty} \equiv \lambda x.\mathsf{R}_0(x)$ is the only function from the empty type into $\mathsf{Tree}(A)$. The following is a formal proof that $\mathsf{leaf}(a_0) \in \mathsf{Tree}(A)$:

$$\dfrac{\dfrac{a \in A \qquad 0 \in \mathsf{N}}{\langle a, 0\rangle \in A \times \mathsf{N}} \qquad \dfrac{[t \in \mathsf{N}_<(0)]_1 \qquad \mathsf{Tree}(A)\ \mathsf{set}}{\mathsf{R}_0(t) \in \mathsf{Tree}(A)}}{\mathsf{sup}(\langle a_0, 0\rangle, \mathsf{R}_0) \in \mathsf{Tree}(A)}\ 1$$

We can build up more complex trees: let

$$e_0(a_0) \equiv \mathsf{leaf}(a_0)$$

then

$$e_1(a_0, a_1) \equiv \mathsf{sup}(\langle a_1, 1\rangle, (x)\ e_0(a_0)) \qquad \begin{array}{c} \bullet\ a_0 \\ | \\ \bullet\ a_1 \end{array}$$

is the tree with the root labeled by a_1 and one child $e_0(a_0)$, and

$$\mathsf{sup}(\langle a_2, 3\rangle, (x)\ \text{if } x \simeq^{\mathsf{N}} 0 \text{ then } e_1(a_0, a_1) \text{ else } e_0(a_0)) \qquad \begin{array}{ccccc} & & \bullet\ a_0 & & \\ & & | & & \\ \bullet\ a_1 & & \bullet\ a_0 & & \bullet\ a_0 \\ & \searrow & | & \swarrow & \\ & & \bullet\ a_2 & & \end{array}$$

is the tree with root $\langle a_2, 3\rangle$ and three sub-trees, the first being $e_1(a_0, a_1)$ and the other two being $e_0(a_0)$.

Let $t \in \mathsf{Tree}(A)$, the root of the tree t and the i-th sub-tree of the tree t are defined by

$$\begin{array}{rcl} \mathsf{root}(t) & \equiv & \mathsf{T}_{rec}(t, (x, y, z)\ x) \\ \mathsf{subtree}(t, i) & \equiv & \mathsf{T}_{rec}(t, (x, y, z)\ y(i)) \end{array}$$

The value of the function $\mathsf{depth}(t) \in \mathsf{N}\ [t : \mathsf{Tree}(A)]$ is the depth of the tree t. Its recursive equation is

$$\mathsf{depth}(\mathsf{sup}(\langle a, n\rangle, b)) = \mathsf{s}(\mathsf{max}(n, (i)\ \mathsf{depth}(b(i))))$$

where the function $\mathsf{max}(n, f) \in \mathsf{N}\ [n : \mathsf{N}, f : (i : \mathsf{N}_<(n))\ \mathsf{N}]$, whose value is the maximum among n arguments, is described in section 6.7.5. The solution is

$$\mathsf{depth}(w) \equiv \mathsf{T}_{rec}(w, (x, y, z)\ \mathsf{s}(\mathsf{max}(\mathsf{snd}(x), z)))$$

A useful function on trees is the function

$$\mathsf{travel}(t) \in \mathsf{Seq}(A)\ [t : \mathsf{Tree}(A)]$$

which associates to a given tree t the sequence of all the labels present in t. It must follow the recursive equations

$$\mathsf{travel}(\mathsf{sup}(\langle a, n\rangle, f)) = a \bullet \mathsf{append}(n, (i)\ \mathsf{travel}(f(i)))$$

where the function

$$\mathsf{append}(n, f) \in \mathsf{Seq}(A)\ [n : \mathsf{N}, f : (i : \mathsf{N}_<(n))\ \mathsf{Seq}(A)]$$

associates to n sequences on A the sequence obtained by appending them; it is defined by solving the following recursive equations

$$\begin{cases} \mathsf{append}(0, f) & = & \mathsf{nil} \\ \mathsf{append}(\mathsf{s}(n), f) & = & \mathsf{append}_2(f(n), \mathsf{append}(n, f)) \end{cases}$$

where $\mathsf{append}_2(s_1, s_2) \in \mathsf{Seq}(A)\ [s_1, s_2 : \mathsf{Seq}(A)]$ is the standard function to append two sequences (see section 6.7.2). Hence we solve them as follow

$$\begin{array}{rcl} \mathsf{append}(n, f) & \equiv & \mathsf{N}_{rec}(n, \mathsf{nil}, (x, y)\ \mathsf{append}_2(f(x), y)) \\ \mathsf{travel}(w) & \equiv & \mathsf{T}_{rec}(w, (x, y, z)\ \mathsf{fst}(x) \bullet \mathsf{append}(\mathsf{snd}(x), z)) \end{array}$$

Supposing $a \in A$ and $t \in \mathsf{Tree}(A)$, we can use the proposition $a\ \mathsf{InSeq}\ s$ to define the proposition $a\ \mathsf{InTree}\ t$ that holds if and only if a is the label of a node in t.

$$a\ \mathsf{InTree}\ t \equiv a\ \mathsf{InSeq}\ \mathsf{travel}(t)$$

Suppose now that $b(x) \in \mathsf{Boole}\ [x : A]$ is a boolean function, and that we are interested to know if in the finite tree t there is a node, whose label is a, such that $b(a) =^{\mathsf{Boole}} \mathsf{true}$. The recursive equation for a solution of this problem is

$$(\mathsf{find}(\mathsf{sup}(\langle a, n\rangle, h), b) = b(a))\ \vee\ \bigvee(n, (i)\ \mathsf{find}(h(i), b))$$

where the function $\bigvee(n, f) \in \mathsf{Boole}\ [n : \mathsf{N}, f : (i : \mathsf{N}_<(n))\ \mathsf{Boole}]$, whose value is the disjunction of n elements, is described in section 6.7.3. The solution of the previous equation is

$$\mathsf{find}(w, b) \equiv \mathsf{T}_{rec}(w, (x, y, z)\ b(\mathsf{fst}(x))\ \vee\ \bigvee(\mathsf{snd}(x), z))$$

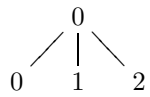
Theorem 6.3.2 *If* $t \in \mathsf{Tree}(A)$, $b(x) \in \mathsf{Boole}\ [x : A]$, *then* $\mathsf{find}(t, b) =^{\mathsf{Boole}} \mathsf{true}$ *if and only if* $(\exists x \in A)\ (f(x) =^{\mathsf{Boole}} \mathsf{true})\ \&\ (x\ \mathsf{InTree}\ t)$.

Proof. The proof is by induction on the construction of t and is similar to the proof of theorem 6.3.1.

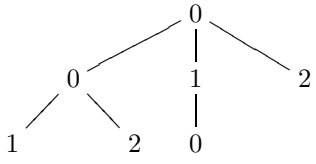
6.3.3 Expanding a finite tree

Suppose that $preds \in A \to \mathsf{Seq}(A)$ and $t \in \mathsf{Tree}(A)$. We want to define a function $\mathsf{expand}(preds, t)$ whose value is the tree t expanded in all leaves, $\mathsf{leaf}(a)$, such that the sub-trees of a are all singleton trees with nodes whose labels are taken from the sequence $preds[a]$. In other words, all leaves $\mathsf{leaf}(a)$ will be replaced by trees with root $\langle a, \sharp preds[a]\rangle$ and sub-trees $\mathsf{leaf}(preds[a]\{i\})$ for $i \in \mathsf{N}_<(\sharp preds\{a\})$. For instance, if $preds \in \mathsf{N} \to \mathsf{Seq}(\mathsf{N})$ maps 0 to the sequence $1 \bullet 2 \bullet \mathsf{nil}$, 1 to the sequence $0 \bullet \mathsf{nil}$ and 2 to the sequence nil, and if $t \in \mathsf{Tree}(\mathsf{N})$ is the tree

$$\mathsf{sup}(\langle 0, 3\rangle, \lambda x.\ \mathsf{leaf}(x))$$



then $\mathsf{expand}(preds, t)$ will be the tree



The following recursive equations holds for the function expand

$$\begin{cases} \mathsf{expand}(preds, \mathsf{sup}(\langle a, 0\rangle, f)) = \\ \qquad \mathsf{sup}(\langle a, \sharp preds[a]\rangle, (i)\ \mathsf{leaf}(preds[a]\{i\})) \\ \mathsf{expand}(preds, \mathsf{sup}(\langle a, \mathsf{s}(n)\rangle, f)) = \\ \qquad \mathsf{sup}(\langle a, \mathsf{s}(n)\rangle, (i)\ \mathsf{expand}(preds, f(i))) \end{cases}$$

This equations can be solved in type theory by

$$\begin{array}{lcll} \mathsf{expand}(preds, t) & \equiv & \mathsf{T}_{rec}(t, (x, y, z) & \text{if } \mathsf{snd}(x) \simeq^{\mathsf{N}} 0 \\ & & & \text{then } \mathsf{sup}(\langle \mathsf{fst}(x), \sharp preds[\mathsf{fst}(x)]\rangle, \\ & & & \qquad (i)\ \mathsf{leaf}(preds[\mathsf{fst}(x)]\{i\})) \\ & & & \text{else } \mathsf{sup}(x, z)) \end{array}$$

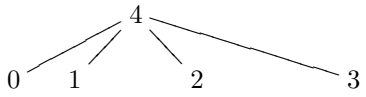
Examples

Define

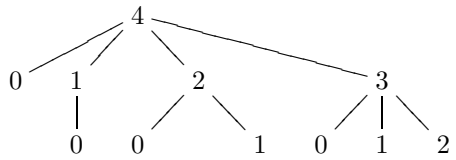
$$\mathsf{pre} \equiv \lambda x.\ \langle x, \lambda y.\ y\rangle \in \mathsf{N} \to \mathsf{Seq}(\mathsf{N}),$$

that is pre is the function which associates to the number x the sequence of all the numbers strictly less then x. Then one can see how some trees can be constructed using repeated expansions:

$\mathsf{expand}(\mathsf{pre}, \mathsf{leaf}(4))$



$(\mathsf{expand}(\mathsf{pre}))^2(\mathsf{leaf}(4))$



and notice that

$$\mathsf{smaller}(k) \equiv (\mathsf{expand}(\mathsf{pre}))^k(\mathsf{leaf}(k))$$

is the tree representing the natural number k together with the ordinary order relation in N.

6.3.4 Finitary Graphs

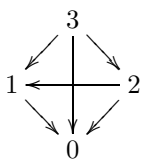
We will identify a finitary graph on A with a function which maps an element to its neighbors:

$$\mathsf{Graph}(A) \equiv A \to \mathsf{Seq}(A)$$

Here are some examples:

$\mathsf{loop} \equiv \lambda x.\ x \bullet \mathsf{nil} \in \mathsf{Graph}(\mathsf{N}_<(1))$

$\mathsf{pre} \in \mathsf{Graph}(\mathsf{N}_<(3))$



A tree t is expanded with respect to a graph g if the children of any node a in the tree are exactly the neighbors of the node a in the graph, i.e. the following equality holds for the predicate Expanded:

$$\mathsf{Expanded}(g, \mathsf{sup}(\langle a, n\rangle, f)) = (\sharp g[a] =^{\mathsf{N}} n)\ \&\ (\forall i < n)\ ((g[a]\{i\} =^{A} \mathsf{root}(f(i)))\ \&\ \mathsf{Expanded}(g, f(i)))$$

This equation can be solved by transfinite recursion.

As an example, there is no tree t such that $\mathsf{Expanded}(\mathsf{loop}, t)$ holds. On the other side, $\mathsf{Expanded}(\mathsf{pre}, \mathsf{smaller}(k))$ is true for any k.

The following sets will be useful:

$$\mathsf{ExpandedTrees}(A, g) \equiv \{t \in \mathsf{Tree}(A) |\ \mathsf{Expanded}(g, t)\}$$

It is the set of all the expanded trees with label in A, while

$$\mathsf{ExpandedTree}(A, g, a) \equiv \{t \in \mathsf{Tree}(A) |\ \mathsf{Expanded}(g, t)\ \&\ \mathsf{root}(t) =^A a\}$$

is the set of all the expanded trees with label in A and root a. The first observation is that there is at most one element in $\mathsf{ExpandedTree}(A, g, a)$.

Theorem 6.3.3 *If A is a set, $g \in \mathsf{Graph}(A)$, $a \in A$ then there is at most one element in $\mathsf{ExpandedTree}(A, g, a)$, i.e. if $t_1, t_2 \in \mathsf{ExpandedTree}(A, g, a)$ then $t_1 = t_2 \in \mathsf{ExpandedTree}(A, g, a)$.*

We are also interested in defining the set of all the trees that are only partially expanded with respect to the graph g, that is their leaves are not fully expanded. We then introduce the proposition

$$\mathsf{PartiallyExpanded}(g, t)\ \mathsf{prop}\ [g : \mathsf{Graph}(A), t : \mathsf{Tree}(A)]$$

which is recursively defined by

$$\begin{aligned}&\mathsf{PartiallyExpanded}(g, \mathsf{sup}(< a, n >, f)) = \\ &\quad (n =^{\mathsf{N}} 0)\ \vee\ ((n =^{\mathsf{N}} \sharp g[a])\ \&\ (\forall i < n)\ ((g[a]\{i\} = \mathsf{root}(f(i)))\ \&\ \mathsf{PartiallyExpanded}(g, f(i))))\end{aligned}$$

As before, this equation can be solved by transfinite recursion. Also the following sets will be useful:

$$\mathsf{PartiallyExpandedTrees}(A, g) \equiv \{t \in \mathsf{Tree}(A) |\ \mathsf{PartiallyExpanded}(g, t)\}$$

It is the set of all the partially expanded trees with label in A, while

$$\mathsf{PartiallyExpandedTree}(A, g, a) \equiv \{t \in \mathsf{Tree}(A) |\ \mathsf{PartiallyExpanded}(g, t) \&\ \mathsf{root}(t) =^A a\}$$

is the set of all the partially expanded trees with label in A and root a. Clearly, each expanded tree is also a partially expanded tree.

We want now to characterized the set $\mathsf{ExpandedTree}(A, g, a)$ as a suitable subset of the set $\mathsf{PartiallyExpandedTrees}(A, g, a)$. First one proves that the function expand, introduced in section 6.3.3, yields a partially expanded tree when applied to a partially expanded tree.

Theorem 6.3.4 *Suppose $g \in \mathsf{Graph}(A)$, $a \in A$ and $t \in \mathsf{PartiallyExpandedTree}(A, g, a)$ Then $\mathsf{expand}(g, t) \in \mathsf{PartiallyExpandedTree}(A, g, a)$.*

Proof. We must prove that

1. $\mathsf{expand}(g, t) \in \mathsf{Tree}(A)$
2. $\mathsf{PartiallyExpanded}(g, \mathsf{expand}(g, t))$ is true
3. $\mathsf{root}(\mathsf{expand}(g, t)) = a$

(1) and (3) immediately follow from the very definition of the function expand. So only (2) must be proved. It will be proved be transfinite induction on the construction of the tree t. Let $t = \mathsf{sup}(\langle a, n\rangle, f)$. The inductive hypothesis is that, for each $i < \sharp g(a)$,

$$\mathsf{expand}(g, f(i)) \in \mathsf{PartiallyExpandedTree}(A, g, \mathsf{root}(f(i)))$$

Two cases must be considered:

- $n = 0$. Then $\mathsf{expand}(g, t) = \mathsf{sup}(\langle a, \sharp g(a)\rangle, (i)\ \mathsf{leaf}(g(a)[i]))$, by definition. If $\sharp g(a) = 0$ then the result is quite immediate, otherwise, if $\sharp g(a) \neq 0$,

  $$\mathsf{PartiallyExpanded}(g, \mathsf{sup}(\langle a, \sharp g(a)\rangle, (i)\ \mathsf{leaf}(g(a)[i])))$$

  is true follows from the fact that $\mathsf{PartiallyExpanded}(g, \mathsf{leaf}(g(a)[i]))$ is true, by the definition of the function leaf.

- $n \neq 0$. Since $t \in \mathsf{PartiallyExpandedTree}(A, g, a)$, then $t = \mathsf{sup}(\langle a, \sharp g(a)\rangle, f)$ and $\sharp g(a) \neq 0$, hence
$$\mathsf{expand}(g, t) = \mathsf{sup}(\langle a, \sharp g(a)\rangle, (i)\ \mathsf{expand}(g, f(i)))$$
and
$$\mathsf{PartiallyExpanded}(g, \mathsf{sup}(\langle a, \sharp g(a)\rangle, (i)\ \mathsf{expand}(g, f(i))))$$
is true follows by using the inductive hypothesis.

Since for each $a \in A$, $\mathsf{leaf}(a) \in \mathsf{PartiallyExpandedTree}(A, g, a)$ we can immediately obtain

Corollary 6.3.5 *Let $n \in \mathsf{N}$, $g \in \mathsf{Graph}(A)$ and $a \in A$. Then*
$$(\mathsf{expand}(g))^n(\mathsf{leaf}(a)) \in \mathsf{PartiallyExpandedTree}(A, g, a).$$

Proof. The statement is a consequence of the previous theorem and the definition of $(\mathsf{expand}(g))^n$.

The main lemma now follows.

Lemma 6.3.6 *If A is a set, $g \in \mathsf{Graph}(A)$, $t \in \mathsf{Tree}(A)$ then*
$$\mathsf{Expanded}(g, t) \text{ if and only if } \mathsf{PartiallyExpanded}(g, t)\ \&\ t = \mathsf{expand}(g, t).$$

Proof. By induction on the construction of t. Let $t \equiv \mathsf{sup}(\langle a, n\rangle, f)$.

- (if) The inductive hypothesis is that, for each $i < n$,
$$\mathsf{Expanded}(g, f(i)) \rightarrow \mathsf{PartiallyExpanded}(g, f(i))\ \&\ f(i) = \mathsf{expand}(g, f(i))$$
Assume $\mathsf{Expanded}(g, \mathsf{sup}(\langle a, n\rangle, f))$ then $\mathsf{PartiallyExpanded}(g, \mathsf{sup}(\langle a, n\rangle, f))$ can be derived as well as
  1. $\sharp g(a) = n$
  2. $\mathsf{Expanded}(g, f(i))$

  It remains to prove
  $$\mathsf{sup}(\langle a, n\rangle, f) = \mathsf{expand}(g, \mathsf{sup}(\langle a, n\rangle, f))$$
  There are two cases to analyze.
  - $n = 0$. By the definition of expand and (1) we have:
  $$\mathsf{expand}(g, \mathsf{sup}(\langle a, n\rangle, f)) = \mathsf{sup}(\langle a, n\rangle, (i)\ \mathsf{leaf}(g(a)[i])) \in \mathsf{Tree}(A)$$
  and since $f(i) = \mathsf{leaf}(g(a)[i]) \in \mathsf{Tree}(A)$ $[i : \mathsf{N}_0]$ can be derived, the result holds in this case.
  - $n \neq 0$. Since (2) holds, by inductive hypothesis we have:
  $$f(i) = \mathsf{expand}(g, f(i)) \in \mathsf{Tree}(A)$$
  and then
  $$\mathsf{sup}(\langle a, n\rangle, f) = \mathsf{sup}(\langle a, n\rangle, (i)\ \mathsf{expand}(g, f(i))) \in \mathsf{Tree}(A)$$
  and, by transitivity, also in this case the result holds.

- (only if) The inductive hypothesis is: for each $i < n$
$$\mathsf{PartiallyExpanded}(g, f(i))\ \&\ f(i) = \mathsf{expand}(g, f(i)) \rightarrow \mathsf{Expanded}(g, f(i))$$
Assume
$$\mathsf{PartiallyExpanded}(g, \mathsf{sup}(\langle a, n\rangle, f))\ \&\ \mathsf{sup}(\langle a, n\rangle, f) = \mathsf{expand}(g, \mathsf{sup}(\langle a, n\rangle, f))$$
From these assumptions we can derive:

1. $\mathsf{sup}(\langle a, n\rangle, f) = \mathsf{expand}(g, \mathsf{sup}(\langle a, n\rangle, f))$
2. $n = \sharp g(a)$
3. $\mathsf{PartiallyExpanded}(g, f(i))$
4. $\mathsf{root}(f(i)) = g(a)[i]$

There are two cases to analyze.

- $n = 0$.
  Since (2) holds, we can easily derive $\mathsf{Expanded}(g, \mathsf{sup}(< \langle a, n\rangle, f))$.
- $n \neq 0$.
  From the definition of the function expand we obtain:

$$\mathsf{expand}(g, \mathsf{sup}(\langle a, n\rangle, f)) = \mathsf{sup}(\langle a, n\rangle, (i)\ \mathsf{expand}(g, f(i))) \in \mathsf{Tree}(A)$$

  which, together with (1), by transitivity gives

$$\mathsf{sup}(\langle a, n\rangle, f) = \mathsf{sup}(\langle a, n\rangle, (i)\ \mathsf{expand}(g, f(i))) \in \mathsf{Tree}(A)$$

  and then $f(i) = \mathsf{expand}(g, f(i))$.
  This and (3) above, by the inductive hypothesis gives $\mathsf{Expanded}(g, f(i))$ which, together with (2) and (4) gives, also in this case, $\mathsf{Expanded}(g, \mathsf{sup}(\langle a, n\rangle, f))$.

We can thus characterize $\mathsf{ExpandedTree}(A, g)$ as the subset of $\mathsf{Tree}(A)$ whose elements are both partially expanded with respect to the graph g and are fixed-points of the function $\mathsf{expand}(g)$.

Theorem 6.3.7 *If A is a set, $a \in A$, $g \in \mathsf{Graph}(A)$ then*

$$\begin{aligned}
\mathsf{ExpandedTree}(A, g) &= \{t \in \mathsf{Tree}(A) |\ \mathsf{PartiallyExpanded}(g, t)\ \&\ (t = \mathsf{expand}(g, t))\} \\
\mathsf{ExpandedTree}(A, g, a) &= \{t \in \mathsf{Tree}(A) |\ \mathsf{PartiallyExpanded}(g, t)\ \&\ (t = \mathsf{expand}(g, t)) \\
&\qquad\qquad \&\ (\mathsf{root}(t) = a)\}
\end{aligned}$$

Proof. The proof is immediate from the previous lemma by using the equal subset formation rule (see [NPS90]).

Remark.
Note that if, for some $m \in \mathsf{N}$, $(\mathsf{expand}(g))^m(\mathsf{leaf}(a)) = (\mathsf{expand}(g))^{m+1}(\mathsf{leaf}(a)) \in \mathsf{Tree}(A)$ then, by corollary 6.3.5 and theorem 6.3.7, $(\mathsf{expand}(g))^m(\mathsf{leaf}(a))$ is the "only" tree in the set $\mathsf{ExpandedTree}(A, g, a)$. We will indeed show that the existence of such an m is also a necessary condition for $\mathsf{ExpandedTree}(A, g, a)$ be inhabited.
Let us prove first a technical lemma.

Lemma 6.3.8 *If A is a set, $a \in A$, $g \in \mathsf{Graph}(A)$ and $\sharp g[a] \neq 0$ then*

$$(\mathsf{expand}(g))^{k+1}(\mathsf{leaf}(a)) = \mathsf{sup}(\langle a, \sharp g[a]\rangle, (i)\ (\mathsf{expand}(g))^k(\mathsf{leaf}(g[a]\{i\})))$$

Proof. By induction on k, the basis case being obvious, let us prove the inductive step:

$$\begin{aligned}
& (\mathsf{expand}(g))^{k+2}(\mathsf{leaf}(a)) && \text{(by inductive hypothesis)} \\
= \ & (\mathsf{expand}(g))((\mathsf{expand}(g))^{k+1}(\mathsf{leaf}(a))) && \text{(by definition of expand)} \\
= \ & \mathsf{expand}(g)(\mathsf{sup}(\langle a, \sharp g(a)\rangle, (i)\ (\mathsf{expand}(g))^k(\mathsf{leaf}(g(a)[i])))) \\
= \ & \mathsf{sup}(\langle a, \sharp g(a)\rangle, (i)\ (\mathsf{expand}(g))((\mathsf{expand}(g))^{k+1}(\mathsf{leaf}(a))))
\end{aligned}$$

Now we have:

Theorem 6.3.9 *If A is a set, $a \in A$, $g \in \mathsf{Graph}(A)$ then*

$$(\forall t \in \mathsf{ExpandedTree}(A, g, a))\ (\exists k \in \mathsf{N})\ t =^{\mathsf{Tree}(A)} (\mathsf{expand}(g))^k(\mathsf{leaf}(\mathsf{root}(t))))$$

Proof. Assume $t \in \mathsf{ExpandedTree}(A, g, a)$. We will prove that

$$(\exists k \in \mathsf{N})\ t =^{\mathsf{Tree}(A)} (\mathsf{expand}(g))^k(\mathsf{leaf}(\mathsf{root}(t))))$$

by induction on the construction of $t = \mathsf{sup}(\langle a, n\rangle, f)$. The inductive hypothesis is:

$$x(i) \in (\exists k \in \mathsf{N})\ f(i) =^{\mathsf{Tree}(A)} (\mathsf{expand}(g))^k(\mathsf{leaf}(\mathsf{root}(f(i))))) \ [i < n]$$

We distinguish two cases.

- $n = 0$. The result immediately follows from:

$$(\mathsf{expand}(g))^0(\mathsf{leaf}(\mathsf{root}(\mathsf{sup}(\langle a, 0\rangle, f)))) = \mathsf{sup}(\langle a, n\rangle, f)$$

- $n \neq 0$. From $\mathsf{sup}(\langle a, n\rangle, f) \in \mathsf{ExpandedTree}(A, g, a)$ we can derive:

  1. $\mathsf{Expanded}(g, \mathsf{sup}(\langle a, n\rangle, f))$
  2. $n = \sharp g(a)$
  3. $\mathsf{root}(f(i)) = g(a)[i]\ [i < n]$
  4. $\mathsf{Expanded}(g, f(i))[i < n]$

  By inductive hypothesis:

$$f(i) = (\mathsf{expand}(g))^{\mathsf{fst}(x(i))}(\mathsf{leaf}(\mathsf{root}(f(i)))) \ [i < n]$$

  hence, from (3):

$$f(i) = (\mathsf{expand}(g))^{\mathsf{fst}(x(i))}(\mathsf{leaf}(g(a)[i])) \ [i < n]$$

  Now, let $x_{\mathsf{max}} = \mathsf{max}(n, (i)\ \mathsf{fst}(x(i)))$, by the theorem 6.3.7, which says that expand acts as the identity on an expanded tree, we have:

$$f(i) = (\mathsf{expand}(g))^{x_{\mathsf{max}}}(\mathsf{leaf}(g(a)[i])) \ [i < n]$$

  moreover, by lemma 6.3.8 since (2) holds, i.e. $n \neq 0$, we have also

$$(\mathsf{expand}(g))^{x_{\mathsf{max}}+1}(\mathsf{leaf}(a)) = \mathsf{sup}(\langle a, n\rangle, (i)\ (\mathsf{expand}(g))^{x_{\mathsf{max}}}(\mathsf{leaf}(g(a)[i])))$$

  and then we obtain

$$(\mathsf{expand}(g))^{x_{\mathsf{max}}+1}(\mathsf{leaf}(a)) = \mathsf{sup}(\langle a, n\rangle, f)$$

  which allows us to conclude:

$$(\exists k \in \mathsf{N})\ (\mathsf{expand}(g))^k(\mathsf{leaf}(a)) = \mathsf{sup}(\langle a, n\rangle, f)$$

Hence

Corollary 6.3.10 *If A is a set, $a \in A$, $g \in \mathsf{Graph}(A)$ and*

$$P(x)\ \mathsf{prop}\ [x : \mathsf{ExpandedTree}(A, g, a)]$$

then

$$(\exists t \in \mathsf{ExpandedTree}(A, g, a))\ P(t)$$

if and only if

$$(\exists m \in \mathsf{N}) \quad (\mathsf{expand}(g))^{m+1}(\mathsf{leaf}(a)) = (\mathsf{expand}(g))^m(\mathsf{leaf}(a)) \\ \&\ P((\mathsf{expand}(g))^m(\mathsf{leaf}(a)))$$

Proof. Let us note that from

$$y \in (\exists t \in \mathsf{ExpandedTree}(A, g, a))\ P(t)$$

we obtain by $\exists$-elimination

1. $\mathsf{fst}(y) \in \mathsf{ExpandedTree}(A, g, a)$
2. $\mathsf{snd}(y) \in P(y_1)$

From (1), by lemma 6.3.6 we obtain $\mathsf{fst}(y) = \mathsf{expand}(g, \mathsf{fst}(y))$, and by theorem 6.3.9 we obtain

$$(\exists m \in \mathsf{N})\ \mathsf{fst}(y) = (\mathsf{expand}(g))^m(\mathsf{leaf}(\mathsf{root}(\mathsf{fst}(y))))$$

Hence

$$(\exists m \in \mathsf{N})\ (\mathsf{expand}(g))^{m+1}(\mathsf{leaf}(a)) = (\mathsf{expand}(g))^m(\mathsf{leaf}(a))\ \&\ P((\mathsf{expand}(g))^m(\mathsf{leaf}(a)))$$

can be derived.

As we noticed in the remark after Theorem 6.3.7, if

$$(\mathsf{expand}(g))^{m+1}(\mathsf{leaf}(a)) = (\mathsf{expand}(g))^m(\mathsf{leaf}(a))$$

then $(\mathsf{expand}(g))^m(\mathsf{leaf}(a))$ is the "only" tree in $\mathsf{ExpandedTree}(A, g, a)$.

6.4 Games and Games trees

6.4.1 Game description

When we want to explain a game to somebody we often start by describing the states (i.e. configurations, situations) of the game. Then we describe the moves of the game, i.e. we describe all the different ways a game can continue from one state to the next. Finally we describe the initial state and describe how to recognize a winning state. If there is only one player that is all, if there are two or more players let us think that the information about who is the player in turn is part of the state of the game and that, for each player, we can explain if a state is winning for him or not. We will consider games which are characterized by the following entities:

$$\langle numPlayers, \mathsf{State}, s_0, \sharp, \mathsf{next}, \mathsf{playerInTurn}, \mathsf{winning}\rangle$$

where

$numPlayers \in \mathsf{N}$	the number of players;
State *set*	the set of states of the game;
$s_0 \in \mathsf{State}$	the initial state;
$\sharp(s) \in \mathsf{N}$ $[s : \mathsf{State}]$	the number of alternative moves in the state s;
$\mathsf{next}(s, i) \in \mathsf{State}$ $[s : \mathsf{State}, i : \mathsf{N}_<(\sharp(s))]$	the state after making the i-th alternative move in the state s;
$\mathsf{winning}(s, k) \in \mathsf{Boole}$ $[s : \mathsf{State}, k : \mathsf{N}_<(numPlayers)]$	is true if the state s is a winning state for the k-th player, false otherwise;
$\mathsf{playerInTurn}(s) \in \mathsf{N}_<(numPlayers)$ $[s : \mathsf{State}]$	the number of the next player to make a move in the state s.

The course of a game starts in an initial state s_0. It proceeds by moving from one state to the next according to the rules of the game. The state of the game is an instantaneous description of the game. For instance, the state of a game played with pieces on a board contains a description of where each piece is situated and which player is in turn. In choosing this way of characterizing a game, we have made the following restrictions:

1. The number of alternative moves in a certain state s is always finite and, moreover, it can be computed from s.

2. All possible next states can be computed from the current state.

3. It is decidable if a state is winning for a player or not.

The reasons for choosing the first two restrictions is that we want to have a general algorithm which decides if it is possible to reach a winning position or not. The third restriction is not a severe restriction; it is just a concrete way of expressing that there should be no doubt if a game is won or not.

6.4.2 Potential Moves

To describe the rules for making a move, it is often more convenient to first describe all potential moves and then disallow some of them as illegal. For instance in chess, we describe how each piece can potentially move and then disallow the illegal ones, for instance it is impossible to move a piece outside the board or to put a piece on already occupied fields. It is often convenient to define the constants State, next, and ♯ in the following way:

- Define a set of "extended states", a set which contains illegal as well as legal states of the game:

$$\mathsf{ExtendedState\ set}$$

- Define, for each extended state s, a boolean which says if s describes a legal state of the game or not:

$$\mathsf{legal}(s) \in \mathsf{Boole}\ [s : \mathsf{ExtendedState}]$$

- Give the upper bound of the number of alternative potential moves in the state s:

$$\mathsf{maxMoves}(s) \in \mathsf{N}\ [s : \mathsf{State}]$$

- Define, for each state s and each potential move i ($0 \leq i < \mathsf{maxMoves}(s)$) the extended state obtained by performing the i-th potential move from s:

$$\mathsf{try}(s, i) \in \mathsf{ExtendedState}\ [s : \mathsf{State}, i : \mathsf{N}_<(\mathsf{maxMoves}(s))]$$

We can show that in this case we can give the description of the game in terms of the entities State, ♯ and next. The states of the game are the subset of the legal extended states, i.e.

$$\mathsf{State} \equiv \{s \in \mathsf{ExtendedState} |\ \mathsf{legal}(s) =^{\mathsf{Boole}} \mathsf{true}\}$$

We can define ♯(s) and $\mathsf{next}(s, i)$ by using a general filtering technique in the following way. We will first build up for each state s the finite sequence of all extended states associated with s:

$$\mathsf{potentialMoves}(s) \equiv \langle \mathsf{maxMoves}(s), \lambda i.\ \mathsf{try}(s, i) \rangle \in \mathsf{Seq}(\mathsf{ExtendedState})$$

Then we will filter out the illegal states, by means of the function filter defined in section 6.3.1.

$$\mathsf{legalMoves}(s) \equiv \mathsf{filter}(\mathsf{potentialMoves}(s), \mathsf{legal})$$

and finally we define

$$\begin{array}{rcl} \sharp(s) & \equiv & \sharp\mathsf{legalMoves}(s) \\ \mathsf{next}(s, i) & \equiv & \mathsf{legalMoves}(s)\{i\} \end{array}$$

6.4.3 The set of game trees.

Let

$$Game \equiv \langle numPlayers, \mathsf{State}, s_0, \sharp, \mathsf{next}, \mathsf{playerInTurn}, \mathsf{winning}\rangle$$

be a given game. We notice that the functions $\sharp$ and next define a graph, namely

$$g_0 \equiv \lambda s.\ \langle \sharp(s), \lambda i.\ \mathsf{next}(s,i)\rangle$$

This is the graph which defines the rules of the game in the sense that the neighbors of any node s in the graph are exactly all possible states immediately following s in the game. In section 6.3.4 we saw how to associate a tree with a graph. For instance, the set

$$\mathsf{ExpandedTree}(\mathsf{State}, g_0, s_0)$$

contains at most one element, the game tree associated with *Game*. This is a tree with a game state in each node and the initial state s_0 in the root. Furthermore, for each node s in the tree, the i-th child of s is the resulting state after choosing the i-th alternative move from s. We saw also, in section 6.3.4, that if there is a game tree, it can be written as $(\mathsf{expand}(g_0))^m(\mathsf{leaf}(s_0))$ for some $m \in \mathsf{N}$.

6.4.4 Some general game problems

Given a game there are many different questions we could ask. For instance, let us consider the following ones:

1. Is it possible for the k-th player to win the game?
2. Is there a winning strategy for the k-th player?
3. Let s be a situation. Give me the list of all the winning moves for the k-th player.

It should be clear that all of these questions can be formalized within type theory by using the set $\mathsf{ExpandedTree}(\mathsf{State}, g_0, s_0)$.

As a first example, let us consider the question: "Is it possible for the k-th player to win the game?" We have to find a boolean function which associates to the k-th player true if and only if there exists the game tree and it has a node labeled with a state which is winning for the k-th player. The problem can be expressed by the following type:

$$(\forall k < numPlayers)\ \{x \in \mathsf{Boole} |\ \ (x = \mathsf{true}) \leftrightarrow (\exists g \in \mathsf{ExpandedTree}(\mathsf{State}, g_0, s_0))\mathsf{HasWinning}(g,k)\}$$

where

$$\mathsf{HasWinning}(g,k) \equiv (\exists a \in \mathsf{State})\ (a\ \mathsf{InTree}\ g)\ \&\ (\mathsf{winning}(a,k) = \mathsf{true}).$$

To solve the second problem: "Is there a winning strategy for the k-th player?", we have to find a boolean function which associates to the k-th player true if and only if there exists the game tree and it is winning for the k-th player. A game tree is winning for the player k if:

1. k is in turn
   - and the situation of the game is winning for k or
   - there exists a move for k to a tree that is winning for k
2. k is not in turn
   - and the situation of the game is winning for k or
   - the situation is not terminal and every move that the player in turn can do turns out in a game tree that is winning for k.

Then the recursive equation defining the proposition

$$\mathsf{IsWinningTree}(t,k)\ [t : \mathsf{Tree}(\mathsf{State}), k < numPlayers]$$

is:

$$\begin{array}{lll}\mathsf{IsWinningTree}(\mathsf{sup}(\langle a,n\rangle,f),k) & = & \text{if } (\mathsf{playerInTurn}(a)=k)\\ & & \text{then } (\mathsf{winning}(a,k)=\mathsf{true})\ \vee\ (\exists i<n)\ \mathsf{IsWinningTree}(f(i),k))\\ & & \text{else } (\mathsf{winning}(a,k)=\mathsf{true})\ \vee ((n\neq 0)\ \&\ (\forall i<n)\ \mathsf{IsWinningTree}(f(i),k))\end{array}$$

This recursive equation can be solved in type theory by a standard technique (see section 6.7.6); then the second problem can be expressed as:

$$\begin{array}{ll}(\forall k<numPlayers)\ \{x\in\mathsf{Boole}| & (x=\mathsf{true})\leftrightarrow\\ & (\exists t\in\mathsf{ExpandedTree}(\mathsf{State},g_0,s_0))\mathsf{IsWinningTree}(t,k)\}\end{array}$$

To solve the problem "Give me the list of all the winning moves for the k-th player in a situation s" we have to find a function which associates to a player k and a state s the sequence of the winning moves for the k-th player from the state s. This is an element w of $\mathsf{Seq}(\mathsf{N}_<(\sharp s))$ which satisfies the condition:

$$(\exists t\in\mathsf{ExpandedTree}(\mathsf{State},g_0,s))(\forall i<\sharp s)\quad (i\ \mathsf{InSeq}\ w)\leftrightarrow\mathsf{IsWinningTree}(\mathsf{subtree}(t,i),k)$$

Now the problem is expressed by the type:

$$\begin{array}{ll}(\forall k<numPlayers)(\forall s\in\mathsf{State})\ \{ & w\in\mathsf{Seq}(\mathsf{N}_<(\sharp s))|\\ & (\exists t\in\mathsf{ExpandedTree}(\mathsf{State},g_0,s))\\ & (\forall i<\sharp s)\ (i\ \mathsf{InSeq}\ w)\leftrightarrow\mathsf{IsWinningTree}(\mathsf{subtree}(t,i),k)\}\end{array}$$

6.4.5 Some general solutions

We can observe that in the types expressing the three considered problems there is always a construction of the form:

$$(\exists t\in\mathsf{ExpandedTree}(\mathsf{State},g_0,s))\ Q(t)$$

for a suitable proposition $Q(t)$ prop $[t : \mathsf{ExpandedTree}(\mathsf{State},g_0,s)]$.

By corollary 6.3.10:

$$\begin{array}{l}(\exists t\in\mathsf{ExpandedTree}(\mathsf{State},g_0,s))\ Q(t)\leftrightarrow\\ \quad(\exists m\in\mathsf{N})\ (\mathsf{expand}(g_0))^{m+1}(\mathsf{leaf}(s))=(\mathsf{expand}(g_0))^m(\mathsf{leaf}(s))\ \&\ Q((\mathsf{expand}(g_0))^m(\mathsf{leaf}(s)))\end{array}$$

holds; hence whenever

$$(\mathsf{expand}(g_0))^{m+1}(\mathsf{leaf}(s))=(\mathsf{expand}(g_0))^m(\mathsf{leaf}(s))$$

holds we can give equivalent formalizations of the problems given in the previous section.

1. $(\forall k<numPlayers)\{x\in\mathsf{Boole}|\ (x=\mathsf{true})\leftrightarrow\mathsf{HasWinning}((\mathsf{expand}(g_0))^m(\mathsf{leaf}(s_0)),k)\}$
2. $(\forall k<numPlayers)\{x\in\mathsf{Boole}|\ (x=\mathsf{true})\leftrightarrow\mathsf{IsWinningTree}((\mathsf{expand}(g_0))^m(\mathsf{leaf}(s_0)),k)\}$
3. $(\forall k<numPlayers)(\forall s\in\mathsf{State})$
   $\{w\in\mathsf{Seq}(\mathsf{N}_<(\sharp s))|\ (\forall i<\sharp s)\ (i\ \mathsf{InSeq}\ w)\leftrightarrow$
   $\mathsf{IsWinningTree}(\mathsf{subtree}((\mathsf{expand}(g_0))^m(\mathsf{leaf}(s)),i),k)\}$

A general solution of the problem (1) is then given by the function

$$\lambda k.\ \mathsf{find}((\mathsf{expand}(g_0))^m(\mathsf{leaf}(s_0)),(s)\ \mathsf{winning}(s,k))$$

since we proved (see theorem 6.3.2) that

$$\mathsf{find}(t,p)\in\{x\in\mathsf{Boole}|\ (x=\mathsf{true})\leftrightarrow(\exists a\in\mathsf{State})\ (a\ \mathsf{InTree}\ g)\ \&\ (p(a)=\mathsf{true})\}$$

For the problem (2) we can easily find a function

$$\mathsf{winningTree}(t,k) \in \mathsf{Boole}\ [t : \mathsf{Tree}(\mathsf{State}), k < numPlayers]$$

satisfying the condition:

$$(\mathsf{winningTree}(t,k) =^{\mathsf{Boole}} \mathsf{true}) \leftrightarrow \mathsf{IsWinningTree}(t,k).$$

Its recursive definition mimics the definition of the type IsWinningTree and can be solved in a similar way (see section 6.7.6). Hence

$$\lambda k.\ \mathsf{winningTree}((\mathsf{expand}(g_0))^m(\mathsf{leaf}(s_0)),k)$$

is a general solution of the problem (2).

Finally, a general solution for the problem (3) is

$$\lambda k.\ \mathsf{filter}((i)\ \mathsf{winningTree}(\mathsf{subtree}(t,i),k), \langle \sharp a, \lambda x.\ x\rangle)$$

whose correctness follows from Theorem 6.3.1.

6.5 Examples

6.5.1 The knight's tour problem

We have an $n \times n$ board with n^2 fields and a knight placed on an initial field. The problem is to find out if there exists a covering of the entire board, i.e. a sequence of moves such that every field of the board is visited exactly once. This is a game with only one player; let us describe it following section 6.4. First, note that there is a fixed number of potential moves: $\mathsf{maxMoves}(s) = 8$, for any state s of the game. A state of this game can be characterized by a current field (the position of the knight) together with the necessary information about the history of the game, i.e., we need to know all the fields visited by the knight before the current one. Clearly, for this game

$$\begin{array}{rcl} numPlayers & \equiv & 1; \\ \mathsf{playerInTurn}(s) & \equiv & 1. \end{array}$$

We represent a field by a pair of natural numbers:

$$\mathsf{Field} \equiv \mathsf{N} \times \mathsf{N}$$

Since we are considering a chess board of a certain dimension n, let us identify the limits of an $n \times n$ board as in fig. 6.1, and, given $f \in \mathsf{Field}$ and $n \in \mathsf{N}$, define the proposition:

$$\mathsf{WithinLimits}(f,n) \equiv (2 \le \mathsf{fst}(f) \le n+1)\ \&\ (2 \le \mathsf{snd}(f) \le n+1)$$

Let us define a field to be busy if it is outside the board or if it has already been visited by the knight; otherwise it is free. A marked-board is a function which associates to each field a boolean saying if the field is free or busy:

$$\mathsf{MarkedBoard} \equiv \mathsf{Field} \to \mathsf{Boole}$$

We will use the abbreviations:

$$\begin{array}{rcl} \mathsf{free} & \equiv & \mathsf{true} \\ \mathsf{busy} & \equiv & \mathsf{false} \end{array}$$

Clearly, in a legal state the current field must be free, but we relax this condition in defining the set:

$$\mathsf{ExtendedState} \equiv \mathsf{Field} \times (\exists n \in \mathsf{N})\ \mathsf{MarkedBoardSize}(n)$$

where

$$\mathsf{MarkedBoardSize}(n) \equiv \{m \in \mathsf{MarkedBoard} |\ (\forall f \in \mathsf{Field})\ \mathsf{Free}(f,m) \to \mathsf{WithinLimits}(f,n)\}$$

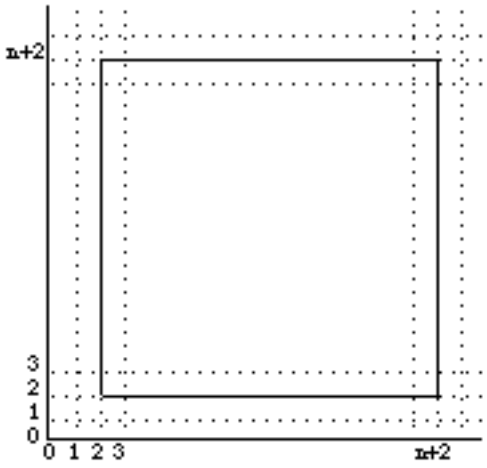


Figure 6.1: the limits of an $n \times n$ board

The elements of this type consist of triples $\langle n, f, m\rangle$ where n is a natural number (the dimension of the board), $f \in \mathsf{Field}$, $m \in \mathsf{MarkedBoardSize}(n)$.

Supposing $s \in \mathsf{ExtendedState}$, we will use the following abbreviations:

$$\begin{array}{rcll} \mathsf{pos}(s) & \equiv & \mathsf{fst}(s) & \text{(the first component of } s \text{ is equal to the actual position)} \\ \mathsf{dim}(s) & \equiv & \mathsf{snd}(s) & \text{(the second component of } s \text{ is equal to the dimension)} \\ \mathsf{occupied}(s) & \equiv & \mathsf{trd}(s) & \text{(the third component of } s \text{ is the marking function)} \end{array}$$

Whenever $f \in \mathsf{Field}$ and $m \in \mathsf{MarkedBoard}$, $\mathsf{Free}(f, m)$ is a proposition which is true if f is free in m. It is defined by:

$$\mathsf{Free}(f, m) \equiv (m[f] =^{\mathsf{Boole}} \mathsf{free})$$

To define the set State as a subset of $\mathsf{ExtendedState}$ we have only to require that the actual field is free; hence:

$$\mathsf{legal}(s) \equiv \mathsf{occupied}(s)[\mathsf{pos}(s)]$$

and

$$\mathsf{State} \equiv \{s \in \mathsf{ExtendedState} |\ (\mathsf{legal}(s) =^{\mathsf{Boole}} \mathsf{true})\}$$

To define $\mathsf{try}(s, i) \in \mathsf{ExtendedState}\ [s : \mathsf{State}, i < \mathsf{maxMoves}(s)]$, let us first define, by using a displacement function according to fig. 6.2,

$$\mathsf{nextField}(s, i) \in \mathsf{Field}\ [s : \mathsf{State}, i < \mathsf{maxMoves}(s)],$$

namely the field reached by choosing the i-th potential move from the state s:

$$\mathsf{nextField}(s, i) \equiv \langle \mathsf{fst}(\mathsf{pos}(s)) + \mathsf{fst}(\mathsf{disp}(i)), \mathsf{snd}(\mathsf{pos}(s)) + \mathsf{snd}(\mathsf{disp}(i))\rangle$$

where

$$\mathsf{disp}(i) \equiv \text{case } i \text{ of } 0 : \langle 2, 1\rangle;\ 1 : \langle 1, 2\rangle;\ \ldots;\ 7 : \langle 2, -1\rangle \text{ endcase}$$

then

$$\mathsf{try}(s, i) \equiv \langle \mathsf{nextField}(s, i), \mathsf{dim}(s), (f) \text{ if } (f = \mathsf{pos}(s)) \text{ then } \mathsf{busy} \text{ else } \mathsf{occupied}(s)[f]\rangle$$

Note that a field is marked as busy only after moving from it.

To complete the description we have to define

$$\mathsf{winning}(s, k) \in \mathsf{Boole}\ [s : \mathsf{State}, k < numPlayers].$$

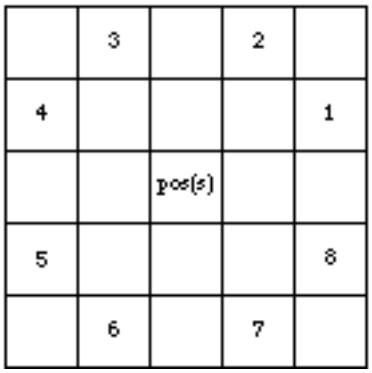


Figure 6.2: the eight potential moves of a knight

We know that a state is winning if there is no free field except for the actual one. This can be checked by filtering a sequence of all the fields within the board:

$$\mathsf{fields}(s) = \langle n^2, \lambda i.\ \langle (i\ \mathsf{div}\ n) + 2, (i\ \mathsf{mod}\ n) + 2\rangle\rangle$$

with the function $(f)\ \mathsf{occupied}(s)[f]$, to obtain the sequence of the free fields in state s:

$$\mathsf{freeFields}(s) \equiv \mathsf{filter}(\mathsf{fields}(s), (f)\ \mathsf{occupied}(s)[f])$$

and then looking if in the filtered sequence there is only the actual field:

$$\mathsf{winning}(s,k) \equiv (\sharp\mathsf{freeFields}(s) = 1)\ \&\ (\mathsf{freeFields}(s)\{0\} = \mathsf{pos}(s)).$$

Hence, (see section 6.4.2) for any initial field s_0, the set of game trees of this game

$$Game \equiv \langle numPlayers, \mathsf{State}, s_0, \sharp, \mathsf{next}, \mathsf{playerInTurn}, \mathsf{winning}\rangle$$

can be defined. Now we have only to note that for $m = n^2 - 1$ (the number of fields in the board minus one), it holds:

$$(\mathsf{expand}(g_0))^{m+1}(\mathsf{leaf}(s_0)) = (\mathsf{expand}(g_0))^{m}(\mathsf{leaf}(s_0))$$

where $g_0 \equiv \lambda s.\ \langle \sharp s, \lambda i.\ \mathsf{next}(s,i)\rangle$. Hence the general solution for the problem (1) developed in section 6.4.5 can be instantiated.

6.5.2 A game with two players

As an example of a game with two players, let us modify the knight's tour in the following way. There is again an $n \times n$ board and a knight placed on an initial field. The moves are performed by two players in turn with the same restrictions as before, i.e. the knight cannot be placed on a yet visited field. A player wins when the opponent cannot move. To describe this game we will use most of the previous definitions. The number of players is 2, hence

$$numPlayers' \equiv 2$$

A state must now comprise the information about whose turn it is, this is achieved simply by adding one more component to the sets ExtendedState and State. Thus:

$$\mathsf{ExtendedState}' \equiv \mathsf{ExtendedState} \times \mathsf{N}_<(numPlayers')$$

and

$$\mathsf{State}' \equiv \{s \in \mathsf{ExtendedState}' |\ \mathsf{legal}'(s) =^{\mathsf{Boole}} \mathsf{true}\}$$

Clearly:

$$\begin{array}{rcl}
\mathsf{pos}'(s) & \equiv & \mathsf{pos}(s) \equiv \mathsf{fst}(s) \\
\mathsf{dim}'(s) & \equiv & \mathsf{dim}(s) \equiv \mathsf{snd}(s) \\
\mathsf{occupied}'(s) & \equiv & \mathsf{occupied}(s) \equiv \mathsf{trd}(s) \\
\mathsf{playerInTurn}'(s) & \equiv & \mathsf{fth}(s) \\
\mathsf{legal}'(s) & \equiv & \mathsf{legal}(s) \equiv \mathsf{occupied}(s)[\mathsf{pos}(s)] \\
\mathsf{nextField}'(s,i) & \equiv & \mathsf{nextField}(s,i)
\end{array}$$

To define $\mathsf{try}'(s,i)$ we first define the function nextPlayer which, given a player j among n-players yields the player in turn after j

$$\mathsf{nextPlayer}(numPlayers, j) \equiv (j+1) \ \mathsf{mod} \ numPlayers$$

then

$$\begin{array}{rl}
\mathsf{try}'(s,i) \equiv \langle & \mathsf{nextField}'(s,i), \\
& \mathsf{dim}'(s), \\
& (f) \text{ if } (f = \mathsf{pos}'(s)) \text{ then busy else } \mathsf{occupied}'(s)(f), \\
& \mathsf{nextPlayer}(numPlayers', \mathsf{playerInTurn}(s))\rangle
\end{array}$$

Finally:

$$\begin{array}{rl}
\mathsf{winning}'(s,k) \equiv & (\mathsf{playerInTurn}'(s) = \mathsf{nextPlayer}(numPlayers', k)) \\
& \& \ \bigwedge(\mathsf{maxMoves}(s), (i) \ \mathsf{occupied}'(s)(\mathsf{nextField}'(s,i)))
\end{array}$$

As before the general solution developed in section 6.4.5 can be instantiated.

6.6 Generality

The ideas and the definitions we will give in this section are not related to a particular paragraph in the chapter but are of general interest.

In general, the function f recursively defined on the index $x \in \mathsf{N}$ by

$$\left\{ \begin{array}{lcl}
f(0,y) & = & k(y) \\
f(\mathsf{s}(x),y) & = & g(x,y,f(x,d(y)))
\end{array} \right.$$

can be solved in type theory by the definition

$$f(x,y) \equiv F(x)[y]$$

where $F(x) \equiv \mathsf{N}_{rec}(x, \lambda y.\ k(y), (u,v)\ \lambda y.\ g(x,y,v(d(y))))$.

The functions f and h recursively defined on $x \in \mathsf{N}$ by

$$\left\{ \begin{array}{lcl}
f(0) & = & k_1 \\
f(\mathsf{s}(x)) & = & g_1(x, f(x), h(x)) \\
h(0) & = & k_2 \\
h(\mathsf{s}(x)) & = & g_2(x, f(x), h(x))
\end{array} \right.$$

can be solved in type theory by

$$\begin{array}{rcl}
f(x) & \equiv & \mathsf{fst}(\mathsf{N}_{rec}(x, \langle k_1, k_2\rangle, (u,v)\ \langle g_1(u, \mathsf{fst}(v), \mathsf{snd}(v)), g_2(u, \mathsf{fst}(v), \mathsf{snd}(v))\rangle)) \\
h(x) & \equiv & \mathsf{snd}(\mathsf{N}_{rec}(x, \langle k_1, k_2\rangle, (u,v)\ \langle g_1(u, \mathsf{fst}(v), \mathsf{snd}(v)), g_2(u, \mathsf{fst}(v), \mathsf{snd}(v))\rangle))
\end{array}$$

In a similar way the function f recursively defined on the set of well-ordering by

$$f(\mathsf{sup}(a,b), k) = g(a,b,k,(v)\ f(b(v), d(k)))$$

can be solved in type theory by

$$f(t,k) \equiv F(t)[k]$$

where $F(t) \equiv \mathsf{T}_{rec}(t, (x,y,z)\ \lambda k.\ g(x,y,k,(v)\ z(v)[d(k)]))$ and the functions f and h recursively defined by

$$\left\{ \begin{array}{lcl}
f(\mathsf{sup}(a,b)) & = & g_1(a,b,(v)\ f(b(v)), (t)\ h(b(t))) \\
h(\mathsf{sup}(a,b)) & = & g_2(a,b,(v)\ f(b(v)), (t)\ h(b(t)))
\end{array} \right.$$

can be solved in type theory by

$$\begin{cases} f(c) & \equiv & \mathsf{fst}(F(c)) \\ h(c) & \equiv & \mathsf{snd}(F(c)) \end{cases}$$

where

$$F(c) \equiv \mathsf{T}_{rec}(c, (x,y,z)\ \langle\ g_1(x,y,(v)\ \mathsf{fst}(z(y(v))), (t)\ \mathsf{snd}(z(y(t)))),\\ g_2(x,y,(v)\ \mathsf{fst}(z(y(v))), (t)\ \mathsf{snd}(z(y(t))))\rangle)$$

In the chapter we will use the following abbreviating definitions:

$$\begin{array}{rclcrcl} \mathsf{R}_0(x) & \equiv & \mathsf{Rec}_0(x) & & \text{if } x \text{ then } y \text{ else } z & \equiv & \mathsf{Rec}_2(x,z,y) \\ \mathsf{fst}(x) & \equiv & \mathsf{E}(x,(u,v)\ u) & & \mathsf{snd}(x) & \equiv & \mathsf{E}(x,(u,v)\ v) \\ \bot & \equiv & \emptyset & & \neg A & \equiv & A \to \bot \\ a =^A b & \equiv & \mathsf{Eq}(A,a,b) & & a \neq^A b & \equiv & \neg\mathsf{Eq}(A,a,b) \end{array}$$

where we omit the superscript A when the type is clear from the context.

6.7 Some type and functions we use

6.7.1 The type $\mathsf{N}_<(a)$

Let $a \in \mathsf{N}$, we want to define a new type $\mathsf{N}_<(a)$ whose canonical elements are all the natural numbers less than a, i.e. $0, \ldots, a-1$; moreover we want that $\mathsf{N}_<(0) = \emptyset$ holds. This can be accomplished by solving the following equations:

$$\begin{cases} \mathsf{N}_<(0) & = & \emptyset \\ \mathsf{N}_<(\mathsf{s}(a)) & = & \{y \in \mathsf{N} |\ y < \mathsf{s}(a)\} \end{cases}$$

where the proposition $x <^{\mathsf{N}} a$, if $x \in \mathsf{N}$ and $a \in \mathsf{N}$, defined by

$$x <^{\mathsf{N}} a \equiv \mathsf{N}_{rec}(a, \emptyset, (u,v)\ (x =^{\mathsf{N}} u)\ \vee\ v)$$

is the type resulting from the solution of the following equations:

$$\begin{cases} x <^{\mathsf{N}} 0 & = & \emptyset \\ x <^{\mathsf{N}} \mathsf{s}(a) & = & (x =^{\mathsf{N}} a)\ \vee\ (x <^{\mathsf{N}} a) \end{cases}$$

Hence,

$$\mathsf{N}_<(a) \equiv \mathsf{N}_{rec}(a, \emptyset, (t,z)\ \{y \in \mathsf{N} |\ y <^{\mathsf{N}} \mathsf{s}(t)\})$$

Let us give you some examples

$$\begin{array}{rcl} \mathsf{N}_<(0) & = & \mathsf{N}_{rec}(0, \emptyset, (t,z)\ \{y \in \mathsf{N} |\ y <^{\mathsf{N}} \mathsf{s}(t)\}) \\ & = & \emptyset \end{array}$$

$$\begin{array}{rcl} \mathsf{N}_<(2) & = & \mathsf{N}_{rec}(2, \emptyset, (t,z)\ \{y \in \mathsf{N} |\ y <^{\mathsf{N}} \mathsf{s}(t)\}) \\ & = & \{y \in \mathsf{N} |\ y <^{\mathsf{N}} 2\} \\ & = & \{y \in \mathsf{N} |\ \mathsf{N}_{rec}(2, \emptyset, (u,v)\ (y =^{\mathsf{N}} u)\ \vee\ v)\} \\ & = & \{y \in \mathsf{N} |\ (y =^{\mathsf{N}} 1)\ \vee\ (y <^{\mathsf{N}} 1)\} \\ & = & \{y \in \mathsf{N} |\ (y =^{\mathsf{N}} 1)\ \vee\ (y =^{\mathsf{N}} 0)\ \vee\ \bot\} \end{array}$$

In the chapter we have used both the notation $(\forall i \in \mathsf{N}_<(n))\ B(i)$ and $(\forall i < n)\ B(i)$ to mean the type $\Pi(\mathsf{N}_<(n), B)$ and $(\exists i \in \mathsf{N}_<(n))\ B(i)$ and $(\exists i < n)\ B(i)$ to mean the type $\Sigma(\mathsf{N}_<(n), B)$; moreover, with the same intention, sometime we denoted the assumption of the variable $k \in \mathsf{N}_<(n)$ by $k < n$.

6.7.2 The function $\mathsf{append}_2(s_1, s_2)$

The recursive equations for the function

$$\mathsf{append}_2(s_1, s_2) \in \mathsf{Seq}(A)\ [s_1, s_2 : \mathsf{Seq}(A)]$$

whose value is the sequence obtained by appending the sequence s_2 to the sequence s_1, are

$$\begin{cases} \mathsf{append}_2(\mathsf{nil}, s_2) & = & s_2 \\ \mathsf{append}_2(a \bullet s_1, s_2) & = & a \bullet \mathsf{append}_2(s_1, s_2) \end{cases}$$

that are solved by putting

$$\mathsf{append}_2(s_1, s_2) \equiv \mathsf{List}_{rec}(s_1, s_2, (x, y, z)\ x \bullet z)$$

6.7.3 The $\bigvee$-function

Let us suppose that $f(x) \in \mathsf{Boole}\ [x : A]$. We want to define the function

$$\bigvee(n, (i)\ f(t(i))) \in \mathsf{Boole}\ [n : \mathsf{N}, t : (x : \mathsf{N})\ A]$$

whose value is true if and only if at least one among $f(t(0)),\ldots,\ f(t(n))$ is true; then the following equations must hold

$$\begin{cases} \bigvee(0, (i)\ f(t(i))) & = & \mathsf{false} \\ \bigvee(\mathsf{s}(n), (i)\ f(t(i))) & = & f(t(\mathsf{s}(n))) \text{ or } \bigvee(n, (i)\ f(t(i))) \end{cases}$$

where x or $y \equiv$ if x then true else y. We can solve them by the definition

$$\bigvee(n, (i)\ f(t(i))) \equiv \mathsf{N}_{rec}(n, \mathsf{false}, (u, v)\ f(t(\mathsf{s}(u))) \text{ or } v).$$

6.7.4 The $\bigwedge$-function

Let us suppose that $f(x) \in \mathsf{Boole}\ [x : A]$. We want to define the function

$$\bigwedge(n, (i)\ f(t(i))) \in \mathsf{Boole}\ [n : \mathsf{N}, t : (x : \mathsf{N})\ A]$$

whose value is true if and only if $f(t(0)),\ldots,\ f(t(n))$ are all true; then the following equations must hold

$$\begin{cases} \bigwedge(0, (i)\ f(t(i))) & = & \mathsf{true} \\ \bigwedge(\mathsf{s}(n), (i)\ f(t(i))) & = & f(t(\mathsf{s}(n))) \text{ and } \bigwedge(n, (i)\ f(t(i))) \end{cases}$$

where x and $y \equiv$ if x then y else false. So

$$\bigwedge(n, (i)\ f(t(i))) \equiv \mathsf{N}_{rec}(n, \mathsf{true}, (u, v)\ f(t(\mathsf{s}(u))) \text{ and } v).$$

6.7.5 The max-function.

We define a function to compute the greatest value among the value in the set $\{f(0), \ldots, f(n)\}$, where $n \in \mathsf{N}$ and $f(x) \in B\ [x : \mathsf{N}]$ where B is a type where a binary order relation $<^B$ is defined. Then we are looking for a function $\mathsf{max}(n, f) \in B\ [n : \mathsf{N}, f : (u : \mathsf{N})\ B]$ such that

$$\begin{cases} \mathsf{max}(0, f) & = & f(0) \\ \mathsf{max}(\mathsf{s}(x), f) & = & \mathsf{max}_2(f(\mathsf{s}(x)), \mathsf{max}(x, f)) \end{cases}$$

and hence

$$\mathsf{max}(n, f) \equiv \mathsf{N}_{rec}(n, f(0), (u, v)\ \mathsf{max}_2(f(\mathsf{s}(u)), v))$$

where $\mathsf{max}_2(x, y) \in B\ [x, y : B]$ is the function to obtain the greatest value among x and y that can be defined by

$$\mathsf{max}_2(x, y) \equiv \text{if } x >^B y \text{ then } x \text{ else } y$$

As an example, we can define the usual order relation among two natural numbers as follow

$$\begin{array}{rcl} x <^{\mathsf{N}} y & \equiv & \mathsf{apply}(\ \mathsf{N}_{rec}(x, \\ & & \lambda t.\ \mathsf{N}_{rec}(t, \mathsf{false}, (u,v)\ \mathsf{true}), \\ & & (u,v)\ \lambda t.\ \mathsf{N}_{rec}(t, \mathsf{false}, (h,k)\ \mathsf{apply}(v, \mathsf{N}_{rec}(t, 0, (n,m)\ n)))), \\ & & y) \end{array}$$

and

$$x \leq^{\mathsf{N}} y \equiv \mathsf{apply}(\mathsf{N}_{rec}(x, \lambda t.\ \mathsf{true}, (u,v)\ \lambda t.\ \mathsf{N}_{rec}(t, \mathsf{false}, (h,k)\ \mathsf{apply}(v, \mathsf{N}_{rec}(t, 0, (n,m)\ n)))), y)$$

Note that in a similar way the usual difference among two natural numbers can be derived (warning: $0 - x = 0 \in \mathsf{N}\ [x : \mathsf{N}]$)

$$x - y \equiv \mathsf{apply}(\mathsf{N}_{rec}(x, \lambda t.\ 0, (u,v)\ \lambda t.\ \mathsf{N}_{rec}(t, \mathsf{s}(u), (h,k)\ \mathsf{apply}(v, \mathsf{N}_{rec}(t, 0, (n,m)\ n)))), y)$$

6.7.6 The sets used in the games description and solutions

Along this section we will use concepts and ideas related with the universes [Mar84, NPS90]. Let $T(x_1, \ldots, x_n)\ \mathsf{set}\ [x_1 : A_1, \ldots, x_n : A_n]$ be the type defined by

$$T(x_1, \ldots, x_n) \equiv t(x_1, \ldots, x_n)$$

where $t(x_1, \ldots, x_n)$ is an element of a universe where the types $A_1, \ldots, A_n$ have a code, constructed by the use of $\&$, $\vee$, $(\forall i < k)\ B(i)$, $(\exists i < k)\ B(i)$, starting from $\bot$ and $a =^{A_i} b$, possibly by using recursion. Then, if for each $1 \leq i \leq n$, one can define the function $x \simeq^{A_i} y \in \mathsf{Boole}\ [x, y : A_i]$ such that $x \simeq^{A_i} y$ has true as a value if and only if $x =^{A_i} y$ is true, it is possible to define a function $f(x_1, \ldots, x_n) \in \mathsf{Boole}\ [x_1 : A_1, \ldots, x_n : A_n]$ such that $f(x_1, \ldots, x_n) =^{\mathsf{Boole}} \mathsf{true}$ if and only if $T(x_1, \ldots, x_n)$ is true.

To obtain f one must only follow the definition of t and use the translation

$$\begin{array}{ll} (\bot)^* = \mathsf{false} & (a =^{A_i} b)^* = a \simeq^{A_i} b \\ (\&)^* = \mathrm{and} & (\vee)^* = \mathrm{or} \\ (\neg)^* = \mathrm{not} & \\ ((\forall i < n)\ B(i))^* = \bigwedge(n, (i)\ (B(i))^*) & (\exists i < n)\ B(i))^* = \bigvee(n, (i)\ (B(i))^*) \end{array}$$

For instance, we can introduce a boolean function $x \simeq^{\mathsf{N}} y \in \mathsf{Boole}\ [x, y : \mathsf{N}]$, whose value is true if and only if x and y have the same value and hence if and only if $x =^{\mathsf{N}} y$, by solving the following recursive equations

$$\begin{cases} 0 \simeq^{\mathsf{N}} 0 & = \mathsf{true} \\ \mathsf{s}(x) \simeq^{\mathsf{N}} 0 & = \mathsf{false} \\ 0 \simeq^{\mathsf{N}} \mathsf{s}(y) & = \mathsf{false} \\ \mathsf{s}(x) \simeq^{\mathsf{N}} \mathsf{s}(y) & = x \simeq^{\mathsf{N}} y \end{cases}$$

The solution, that is obtained by using the ideas of the previous sections, is

$$\begin{array}{l} x \simeq^{\mathsf{N}} y \\ \equiv \mathsf{apply}(\ \mathsf{N}_{rec}(x, \\ \qquad \lambda t.\ \mathsf{N}_{rec}(t, \mathsf{true}, (u,v)\ \mathsf{false}), \\ \qquad (u,v)\ \lambda t.\ \mathsf{N}_{rec}(t, \mathsf{false}, (h,k)\ \mathsf{apply}(v, \mathsf{N}_{rec}(t, 0, (n,m)\ n)))), \\ \qquad y) \end{array}$$

Much easier is to define the boolean function

$$x \simeq^{\mathsf{Boole}} y \in \mathsf{Boole}\ [x, y : \mathsf{Boole}],$$

whose value is true if and only if x and y have the same value:

$$x \simeq^{\mathsf{Boole}} y \equiv \text{if } x \text{ then } y \text{ else } \mathrm{not}(y)$$

where the boolean function

$$\mathrm{not}(x) \in \mathsf{Boole}\ [x : \mathsf{Boole}]$$

is defined by

$$\mathrm{not}(x) \equiv \text{if } x \text{ then } \mathsf{false} \text{ else } \mathsf{true}$$

The set $\mathsf{Expanded}(g,t)$

We must solve in type theory the following equation

$$\begin{array}{l}\mathsf{Expanded}(g,\mathsf{sup}(\langle a,n\rangle,f)) = \\ \quad (\sharp g(a) =^{N} n)\ \&\ (\forall i<n)\ (g(a))[i] =^{\mathsf{Tree}(A)} \mathsf{root}(f(i))\ \&\ \mathsf{Expanded}(g,f(i))\end{array}$$

The solution can be obtained by transfinite induction in a universe U where the type A has a code. In this case we must solve the following equation

$$\begin{array}{l}\mathsf{Expanded}(g,\mathsf{sup}(\langle a,n\rangle,f)) = \\ \quad (\sharp g(a) =^{N} n)\ \&\ \bigwedge(n,(i)\ g(a))[i] =^{\mathsf{Tree}(A)} \mathsf{root}(f(i))\ \&\ \mathsf{Expanded}(g,f(i)))\end{array}$$

and hence to obtain the code for $\mathsf{Expanded}(g,t)$ we make the definition

$$\begin{array}{l}\mathsf{Expanded}(g,t) \equiv \mathsf{T}_{rec}(t,(x,y,z)\ \sharp g(\mathsf{fst}(x)) =^{N} \mathsf{snd}(x)\ \& \\ \qquad \bigwedge(\mathsf{snd}(x),(i)\ g(a))[i] =^{\mathsf{Tree}(A)} \mathsf{root}(y(i))\ \&\ \mathsf{Expanded}(g,y(i)))\end{array}$$

The proposition $\mathsf{IsWinningTree}(t,k)$

Now, let us analyze the definition of the proposition

$$\mathsf{IsWinningTree}(t,k)\ [t:\mathsf{Tree}(\mathsf{State}), k<numPlayers];$$

This proposition must satisfy the equation

$$\begin{array}{l}\mathsf{IsWinningTree}(\mathsf{sup}(\langle a,n\rangle,f),k) = \\ \quad \text{if } (\mathsf{playerInTurn}(a) \simeq^{N} k) \\ \quad \text{then } (\mathsf{winning}(a,k) =^{\mathsf{Boole}} \mathsf{true}) \vee \\ \qquad\qquad (\exists i<n)\ \mathsf{IsWinningTree}(f(i),k) \\ \quad \text{else } (\mathsf{winning}(a,k) =^{\mathsf{Boole}} \mathsf{true}) \vee \\ \qquad\qquad ((n \neq^{N} 0)\ \&\ (\forall i<n)\ \mathsf{IsWinningTree}(f(i),k))\end{array}$$

This equation can be solved by the definition

$$\mathsf{IsWinningTree}(t,k) \equiv (\mathsf{isWinningTree}(t,k) =^{\mathsf{Boole}} \mathsf{true})$$

where

$$\begin{array}{l}\mathsf{isWinningTree}(t,k) \equiv \\ \quad \mathsf{apply}(\ \mathsf{T}_{rec}(\ t, \\ \qquad\qquad (x,y,z)\ \lambda k.\quad \text{if } (\mathsf{playerInTurn}(\mathsf{fst}(x)) \simeq^{N} k) \\ \qquad\qquad\qquad\qquad \text{then } \mathsf{winning}(\mathsf{fst}(x),k) =^{\mathsf{Boole}} \mathsf{true} \\ \qquad\qquad\qquad\qquad\quad \vee(\exists i<n)\ z(i)[k] \\ \qquad\qquad\qquad\qquad \text{else } \mathsf{winning}(\mathsf{fst}(x),k) =^{\mathsf{Boole}} \mathsf{true} \\ \qquad\qquad\qquad\qquad\quad \vee(\neg(\mathsf{snd}(x) =^{\mathsf{N}} 0) \\ \qquad\qquad\qquad\qquad\quad \&(\forall i<n)\ z(i)[k])), \\ \qquad k)\end{array}$$

Then the set $\mathsf{IsWinningTree}(t,k)$ is inhabited if and only if the following function

$$\mathsf{winningTree}(t,k) \in \mathsf{Boole}\ [t:\mathsf{Tree}(\mathsf{State}), k<numPlayers]$$

has true as value:

$$\begin{array}{l}\mathsf{winningTree}(t,k) \equiv \\ \quad \mathsf{apply}(\ \mathsf{T}_{rec}(\ t, \\ \qquad\qquad (x,y,z)\ \lambda k.\quad \text{if } (\mathsf{playerInTurn}(\mathsf{fst}(x)) \simeq^{\mathsf{N}} k) \\ \qquad\qquad\qquad\qquad \text{then } (\mathsf{winning}(\mathsf{fst}(x),k) \simeq^{\mathsf{Boole}} \mathsf{true}) \\ \qquad\qquad\qquad\qquad\quad \text{or } \bigvee(\mathsf{snd}(x),(i)\ z(i)[k]) \\ \qquad\qquad\qquad\qquad \text{else } (\mathsf{winning}(\mathsf{fst}(x),k) \simeq^{\mathsf{Boole}} \mathsf{true}) \\ \qquad\qquad\qquad\qquad\quad \text{or } (\text{not}(\mathsf{snd}(x) \simeq^{\mathsf{N}} 0) \text{ and} \\ \qquad\qquad\qquad\qquad\quad \bigwedge(\mathsf{snd}(x),(i)\ z(i)[k]))), \\ \qquad k)\end{array}$$

Chapter 7

What should be avoided

7.1 Summary

In this chapter we will analyze an extension of Martin-Löf's intensional type theory by means of a set constructor $\mathcal{P}$ such that the elements of $\mathcal{P}(S)$ are the subsets of the set S.

Since it seems natural to require some kind of extensionality on the equality among subsets, it turns out that such an extension cannot be constructive. In fact we will prove that this extension is classic, that is $(A \vee \neg A)$ true holds for any proposition A.

7.2 Introduction

In [GR94] it is shown that the proof theoretic strength of Martin-Löf's set theory [Mar84, NPS90] with restricted well-orders and the universe of the small sets is that of a subsystem of second order arithmetic with Δ^1_2 comprehension and bar-induction. Thus, it is natural to wonder whether it is possible to enforce it to a theory with the strength of the full comprehension schema by adding a power-set constructor; in fact, this extension is necessary if we want to quantify over the subsets of a given set since in Martin-Löf's set theory quantification is meaningful only on elements of a set.

In the literature there are already examples of constructive set theories with some kind of power-set constructor. For instance, one can think of a *topos* as a "generalized set theory" by associating with any topos its internal language (see [Bel88]). The logic underlying such a set theory is the intuitionistic predicate calculus and so any topos can be thought of as an intuitionistic universe of sets. Then, the lack of the rule of excluded middle seems to assure the constructivity of any proof developed within topos theory. The problem of adapting the topos theoretic approach to Martin-Löf's set theory is due to the impredicativity of the former. Indeed, Martin-Löf's set theory is predicative and provides a fully algorithmic way to construct the elements of the sets and the proofs of the propositions over these sets.

Another approach which should be considered is the Calculus of Constructions by Coquand and Huet, where the power of a set S can be identified with the collection of the functions from S into prop. But, if we identify sets and propositions, which is basic for a constructive explanation of the meaning of Martin-Löf's set theory, the power-set so obtained is *not* a set, since prop cannot be a set and hence also the collection of the functions from a set S to prop cannot be a set. Thus, there is no chance to give a constructive, i.e. intuitionistic and predicative, meaning to quantification over its elements. A second problem with this approach is that in this way we would obtain an intensional notion of power-set, which is not the intended one since we think that equality among subsets has to be understood extensionally. Finally, it can be proved that the strong sum type, which is characteristic in Martin-Löf's set theory, cannot consistently be added to the Calculus of Constructions at the level of propositions (see [Coq90]); thus, this approach cannot have the full strong sum on propositions (see for instance [Luo90]) and hence it cannot be considered an extension of Martin-Löf's set theory.

Of course, there is no reason to expect that a second order construction becomes constructive

only because it is added to a theory which is constructive. Indeed, we will prove that even the weaker fragment iTT, which contains only the basic set constructors, i.e. no universes and no well-orders, and the *intensional* equality, cannot be extended with a power-set constructor in a way compatible with the usual semantical explanation of the connectives, if the power-set is the collection of all the subsets of a given set equipped with extensional equality expressed in a uniform way at the propositional level. In fact, by using the so called intuitionistic axiom of choice, it is possible to prove that, given any power-set constructor, which satisfies the conditions that we will illustrate in the next section, classical logic arises (see also [Hof95] page 170, where it is suggested that a similar result holds in the setoid model built upon the Calculus of Constructions). A crucial point in carrying on our proof is the uniformity of the equality condition expressing extensionality on the power-set. This is to be contrasted with the proofs of similar results already proposed in the literature, after Diaconescu's original proof in [Dia75], where proof-irrelevance of propositions, which does not hold in constructive type theory, is used.

7.3 $\mathsf{iTT}^P = \mathsf{iTT}$ + power-sets

To express the rules and the conditions that we are going to require on the power-set we need to use judgements of the form A true (see [Mar84]) and hence it is convenient to recall their main property: A true holds if and only if there exists a proof-element a such that $a \in A$ holds (for a formal approach to this topic see [Val98]). In particular, the following rule is admissible

$$\text{(True Introduction)} \quad \frac{a \in A}{A \text{ true}}$$

as well as all the rules of the intuitionistic predicative calculus with equality, where the judgement A true is the type theoretic interpretation of $\vdash A$ (see [Mar84] for the definition of the embedding of the intuitionistic predicative calculus within iTT). Here, we only recall the rules for the set of the intensional propositional equality Id (see [NPS90], page 61) which plays a central role in this chapter (for sake of clearness, supposing A is a set and $a, b \in A$, we will write $a =_A b$ to mean $\mathsf{Id}(A, a, b)$). The formation and introduction rules are

$$\frac{A \text{ set} \quad a \in A \quad b \in A}{a =_A b \text{ set}} \qquad \frac{A = C \quad a = c \in A \quad b = d \in A}{(a =_A b) \;=\; (c =_A d)}$$

$$\frac{A \text{ set} \quad a \in A}{\mathsf{r}(a) \in a =_A a} \qquad \frac{A \text{ set} \quad a = b \in A}{\mathsf{r}(a) = \mathsf{r}(b) \in a =_A a}$$

whereas the elimination rule is

$$\frac{\begin{array}{cc} & [x : A]_1 \\ & \vdots \\ c \in a =_A b \quad d(x) \in C(x, x, \mathsf{r}(x)) & C(x, y, z) \text{ set} \end{array} \qquad \begin{array}{c} [x : A, y : A, z : x =_A y]_1 \\ \vdots \\ \end{array}}{\mathsf{K}(c, d) \in C(a, b, c)}\ 1$$

and, if $C(x, y, z)$ set $[x : A, y : A, z : x =_A y]$ and $D(x, y)$ set $[x : A, y : A]$, it yields the admissibility of the following two rules:

$$\frac{\begin{array}{cc} & [x : A] \\ & \vdots \\ c \in a =_A b \quad C(x, x, \mathsf{r}(x)) \text{ true} & \end{array}}{C(a, b, c) \text{ true}} \qquad \frac{\begin{array}{c} [x : A] \\ \vdots \\ a =_A b \text{ true} \quad D(x, x) \text{ true} \end{array}}{D(a, b) \text{ true}}$$

The rules for the set $\mathcal{P}(S)$ depend on the definition of what a subset is within iTT. Following a long tradition, we identify a subset of S with a propositional function on S, i.e., provided that $U(x)$ set $[x : S]$, we say that $U \equiv (x : S)\, U(x)$ is a subset of S, and hence, we say that an element $a \in S$ is an element of U if $U(a)$ is inhabited, i.e. the judgement $U(a)$ true holds (see [Bru80] and [SV98] for a detailed discussion on this topic).

Thus, provided that we want to have an extensional equality between subsets, we are forced to consider equal two subsets U and V of S if and only if they have the same elements, i.e. $U(x) \leftrightarrow V(x)$ true $[x : S]$.

The will to construct a set out of the collection of the propositional functions over a set equipped with an equality relation between propositional functions based on equi-provability is the point where classical logic breaks into the system.

Inspired by the previous explanations, here we propose the following formation and introduction rules for $\mathcal{P}(S)$:

Formation

$$\frac{S \text{ set}}{\mathcal{P}(S) \text{ set}} \qquad \frac{S = T}{\mathcal{P}(S) = \mathcal{P}(T)}$$

Introduction

$$\frac{U(x) \text{ set } [x : S]}{\{(x : S)\ U(x)\} \in \mathcal{P}(S)}$$

Now, we should formulate the next rules for the set $\mathcal{P}(S)$, i.e. the equality introduction rule, the elimination rule and the equality rule. But the aim of this chapter is to show that it is actually impossible to formulate any rules which make valid the conditions that we are going to discuss in the following and that seem to be necessary to make $\mathcal{P}(S)$ the power-set of S, because otherwise we would obtain a Heyting semantics for classical logic.

As already said, it is necessary to formalize the fact that the equality between subsets is extensional; otherwise, $\mathcal{P}(S)$ would not be the set of the subsets of S but the collection of the propositional functions over S, and to add this collection as a set is not consistent (see [Jac89]). Thus, one seems to be forced to require that, whenever the two subsets U and V of S are equal, that is if $U(x) \leftrightarrow V(x)$ true $[x : S]$ then $\{(x : S)\ U(x)\} = \{(x : S)\ V(x)\} \in \mathcal{P}(S)$. However, as noted by Peter Aczel after reading a preliminary version of this work, this should not be a formal rule for the set $\mathcal{P}(S)$ since the use of an extensional equality rule for power-sets does not fit with the idea of treating the judgmental equalities as definitional, which is basic in iTT. To avoid this problem, we require here a weaker condition, which is a consequence of the judgmental equality above.

Equality introduction condition
Let $U(x) \leftrightarrow V(x)$ true $[x : S]$. Then there exists a proof-term $\mathsf{c}(U, V)$ such that

$$\mathsf{c}(U, V) \in \{(x : S)\ U(x)\} =_{\mathcal{P}(S)} \{(x : S)\ V(x)\}$$

Also this condition does not follow completely the general approach used in Martin-Löf's set theory since some information is lost in the path from the premise to the conclusion, that is the proof term which testifies that

$$U(x) \leftrightarrow V(x) \text{ true } [x : S]$$

For this reason we do not want to consider it a formal rule. In the following we will prove that this lack of information is one of the main point in obtaining classical logic by adding the power-set constructor and this fact suggests that there is still some hope to be able to add a power-set constructor to constructive set-theory. For instance, one could consider the following rule

$$\frac{f(x) \in U(x) \leftrightarrow V(x)\ [x : S]}{\mathsf{c}(U, V, f) \in \{(x : S)\ U(x)\} =_{\mathcal{P}(S)} \{(x : S)\ V(x)\}}$$

and in this case it would be no more possible to carry on our proof to its end. In any case it is worth noting that even this approach is not sufficient to avoid classical logic in most of the actual implementations of constructive set theory (see for instance [Mag92]). Indeed, such implementations use pattern-matching instead of elimination rules and thus they validate stronger conditions, as the uniqueness of equality proofs [HS95] which allows to obtain classical logic also with this rule if an extensional power-set would be added. Moreover, this rule seems not to satisfy the condition that any element in a propositional equality set is of the form $\mathsf{r}(-)$, i.e. we loose the adequacy property of the calculus.

Before going on, it is worth noting that the previous equality condition, even if reasonable, can already be pretty dangerous. In fact, if we try to follow the same approach within a calculus which allows to quantify over the collection of all the propositions, then we would obtain *proof-unicity*. And note that to allow to quantify over the collection of all the propositions is not really different than to allow to quantify over the collection $\mathcal{P}(\top)$ of the subsets of the one-element set $\top$.

To prove this result first observe that the following lemma holds.

Lemma 7.3.1 *The intensional equality proposition is equivalent to the Leibniz equality proposition.*

Proof. Recall that the Leibniz equality is defined as follows by using a quantification over the collection of the propositional functions.

$$\mathsf{LEq}(A, a, b) \equiv (\forall P \in A \to \mathsf{prop})\ P(a) \to P(b)$$

that is, *those things which cannot be distinguished by means of a proposition are equal*.

Now, we want to show that, for any set A, and elements a and b in A, $a =_A b$ if and only if $\mathsf{LEq}(A, a, b)$. Thus, let us suppose that $c \in a =_A b$, $P : A \to \mathsf{prop}$, $w : P(a)$, and put $Q(x, y, z) \equiv P(x) \to P(y)$. Then, supposing x is any element in A, $Q(x, x, \mathsf{r}(x)) \equiv P(x) \to P(x)$ and hence, $\lambda((y)\ y) \in Q(x, x, \mathsf{r}(x))$. So, by using the elimination rule for intensional equality, we obtain $\mathsf{K}(c, \lambda((y)\ y)) \in P(a) \to P(b)$ and hence, by discharging the assumptions $P : A \to \mathsf{prop}$ and $w : P(a)$,

$$\lambda((P)\ \lambda((w)\ \mathsf{K}(c, \lambda((y)\ y))[w])) \in \mathsf{LEq}(A, a, b)$$

On the other hand, suppose $c \in \mathsf{LEq}(A, a, b)$, that is

$$c \in (\forall P \in A \to \mathsf{prop})\ P(a) \to P(b),$$

and put $P \equiv (x : A)\ a =_A x$. Then $c[P] \in (a =_A a) \to (a =_A b)$. But we know that $\mathsf{r}(a) \in a =_A a$ and hence

$$c[P][\mathsf{r}(a)] \in a =_A b$$

Now, we can prove the following theorem.

Theorem 7.3.2 *Extensionality yields proof-unicity, that is, if*

$$\textit{(extensionality)} \quad (\forall P, Q \in \mathsf{prop})\ (P \leftrightarrow Q) \to P =_{\mathsf{prop}} Q$$

then

$$\textit{(proof-unicity)} \quad (\forall P \in \mathsf{prop})(\forall x, y \in P)\ x =_P y$$

Proof. Let P be any proposition and put $Q \equiv \top$, that is Q is the one-proof proposition inductively generated by the following introduction and elimination rules.

$$* \in \top \qquad \frac{c \in \top \qquad C(x)\ \mathsf{prop}\ [x : \top] \qquad d \in C(*)}{\mathsf{R}_1(c, d) \in C(c)}$$

Suppose now that x and y are proofs of P. Then P is true and hence $P \leftrightarrow \top$ holds (for instance one can consider the proof-elements $\lambda((z)\ *) \in P \to \top$ and $\lambda((w)\ x) \in \top \to P$). Then extensionality yields $P =_{\mathsf{prop}} \top$. Consider now the following property on propositions:

$$\mathsf{OneEl}(A) \equiv (\forall x, y \in A)\ x =_A y$$

which states that the proposition A has at most one proof-element. It is easy to show that $\mathsf{OneEl}(\top)$ holds. In fact, let us put $C(x) \equiv x =_\top *$ and use the $\top$-elimination rule in order to obtain, for any x and y in $\top$, that $\mathsf{R}_1(x, \mathsf{r}(*)) \in x =_\top *$ and $\mathsf{R}_1(y, \mathsf{r}(*)) \in y =_\top *$. Hence it is sufficient to use the fact that for the intensional equality proposition symmetry and transitivity hold to obtain that $x =_\top y$ holds.

But then $P =_{\mathsf{prop}} \top$ yields, by the previous lemma, that P and $\top$ satisfy the same propositions and hence also $\mathsf{OneEl}(P)$ holds, that is, we obtained proof-unicity.

Now we have an immediate corollary.

Corollary 7.3.3 *Extensionality is not consistent with inductive propositions and strong elimination*[1].

Proof. The proof is straightforward since by using strong elimination it is possible to prove that there are inductive propositions with more then one proof (see next lemma 7.5.2 for a proof which does not use strong elimination).

The elimination and the equality rules are even more problematic. In fact it is difficult to give a plain application of the standard approach that requires to obtain the elimination rule out of the introduction rule(s) (see [Mar71]). In fact, the introduction rule does not act over elements of a set but over elements of the collection $((x : S)\ \mathsf{set})_{\leftrightarrow}$. Thus, if one wants to follow for $\mathcal{P}(S)$ the general pattern for a quotient set, he could look for a rule similar to the following:

$$\frac{\begin{array}{cccc} & [Y : (x : S)\ \mathsf{set}] & & [Y, Z : (x : S)\ \mathsf{set}, Y(x) \leftrightarrow Z(x)\ \mathsf{true}\ [x : S]] \\ & \vert & & \vert \\ c \in \mathcal{P}(S) & d(Y) \in C(\{Y\}) & & d(Y) = d(Z) \in C(\{Y\}) \end{array}}{\mathsf{P}_{\mathsf{rec}}(c, d) \in C(c)}$$

But this rule requires the use of variables for propositional functions, which are difficult to justify since prop is not a set.

Moreover, a standard equality rule should be something similar to the following

$$\frac{\begin{array}{ccccc} [x : S] & & [Y : (x : S)\ \mathsf{set}] & & [Y, Z : (x : S)\ \mathsf{set}, Y(x) \leftrightarrow Z(x)\ \mathsf{true}\ [x : S]] \\ \vert & & \vert & & \vert \\ U(x)\ \mathsf{set} & & d(Y) \in C(\{Y\}) & & d(Y) = d(Z) \in C(\{Y\}) \end{array}}{\mathsf{P}_{\mathsf{rec}}(\{(x : S)\ U(x)\}, d) = d((x : S)\ U(x)) \in C(\{(x : S)\ U(x)\})}$$

These rules are a direct consequence of the introduction rule and the equality introduction condition and they are already not completely within standard Martin-Löf's set theory. But, the problem is that, as they stand, they are not sufficient to make $\mathcal{P}(S)$ the set of the subsets of S. For instance, there is no way to obtain a proposition out of an element of $\mathcal{P}(S)$ and this does not fit with the introduction rule. Thus, to deal with the set $\mathcal{P}(S)$, one should add some rules which links its elements both with the elements of the type set and with those of the collection $\mathsf{set}_{\leftrightarrow}$, whose elements are propositions but whose equality is induced by the logical equivalence.

Again, we don't want to propose any particular rule since we are going to show that there can be no suitable rule, but we simply require that two conditions, which should be a consequence of such rules, are satisfied. The first condition is:

Elimination condition
Let $c \in \mathcal{P}(S)$ and $a \in S$. Then there exists a proposition $a \varepsilon c$.

This condition is suggested by the elimination rule that we have considered. In fact, a free use of the elimination rule with $C(z) \equiv \mathsf{set}_{\leftrightarrow}$ allows to obtain that $\mathsf{P}_{\mathsf{rec}}(c, (Y)\ Y(a))$ is an element of $\mathsf{set}_{\leftrightarrow}$ and hence that it is a proposition and we can identify such a proposition with $a \varepsilon c$. Of course, the above condition is problematic because it requires the existence of a proposition but it gives no knowledge about it; in particular it is not clear if one has to require a new proposition (which are its canonical elements? which are its introduction and elimination rules?) or an old one (which proposition should one choose?).

As a consequence of the suggested equality rule, we require the following equality condition.

Equality condition
Suppose $U(x)\ \mathsf{set}\ [x : S]$ and $a : S$ then $a \varepsilon \{(x : S)\ U(x)\} \leftrightarrow U(a)\ \mathsf{true}$.

This condition can be justified in a way similar to the justification of the elimination condition, but using the equality rule instead of the elimination rule; in fact, supposing $U(x)\ \mathsf{set}\ [x : S]$ and $a : S$, the equality rule allows to obtain that $a \varepsilon \{(x : S)\ U(x)\}$ and $U(a)$ are equal elements of $\mathsf{set}_{\leftrightarrow}$ which yields our condition. This condition cannot be justified from a semantical point of view since

[1] Strong elimination corresponds to the elimination rule for the universe U in the appendix B.

we have no way to recover the proof element for its conclusion; this is the requirement which allows us to develop our proof in the next section without furnishing term constructors for classical logic.

It is worth noting that no form of η-equality, like

$$\frac{c \in \mathcal{P}(S)}{\{(x : S)\ x\varepsilon c\} = c \in \mathcal{P}(S)} \quad x \notin VF(c),$$

is required on $\mathcal{P}(S)$, but its validity is a consequence of the suggested elimination rule for $\mathcal{P}(S)$ at least within the extensional version of Martin-Löf's set theory eTT. This theory is obtained from iTT by substituting the intensional equality proposition by the extensional equality proposition $\mathsf{Eq}(A, a, b)$ which allows to deduce $a = b \in A$ from a proof of $\mathsf{Eq}(A, a, b)$. The problem with extensional equality is that it causes the lack of decidability of the equality judgement; for this reason it is usually rejected in the present version of the theory. To prove the η-equality in eTT let us assume that Y is a subset of S and that $x : S$, then $Y(x)$ set and hence $x\varepsilon\{Y\} \leftrightarrow Y(x)$ true holds because of the equality condition and it yields $\mathsf{Eq}(\mathcal{P}S, \{(x : S)\ x\varepsilon\{Y\}\}, \{Y\})$; thus, if $c \in \mathcal{P}S$, by using the elimination rule one obtains $\mathsf{Eq}(\mathcal{P}S, \{(x : S)\ x\varepsilon c\}, c)$ and hence $\{(x : S)\ x\varepsilon c\} = c \in \mathcal{P}S$. Note that the last step is not allowed in iTT^P.

It is interesting to note that also this kind of extensionality is dangerous if we are working within a framework which allows to quantify over the collection of all the propositions. In fact in this case we can argue like in [Coq90] (or also [Jac89]) where it is proved that if there exists a proposition B and two elements $\mathsf{code} \in \mathsf{prop} \to B$ and $\mathsf{decode} \in B \to \mathsf{prop}$ such that

$$(\forall A \in \mathsf{prop})\ A \leftrightarrow \mathsf{decode}(\mathsf{code}(A))$$

then we obtain an inconsistent theory since the inconsistent PTS λ_U (see [Bar92] for its definition) can be embedded into it.

Now, note that the possibility to build the power-set of the one-element set, together with the required logical equivalence condition for the membership proposition, allow us to define a set B and the necessary elements by putting

$$\begin{aligned} &B \equiv \mathcal{P}(\top) \\ &\mathsf{code}(U) \equiv \{w \in \top |\ U\} \\ &\mathsf{decode}(c) \equiv *\varepsilon c \end{aligned}$$

In fact, the required equivalence condition yields that, for any proposition U,

$$*\varepsilon\{w \in \top |\ U\} \leftrightarrow U$$

that is

$$\mathsf{decode}(\mathsf{code}(U)) \leftrightarrow U$$

Then it is clear that we have to avoid to use this kind of power-set constructor together with impredicative quantification.

7.4 iTT^P is consistent

It is well known that by adding as a set to iTT the collection $\mathcal{P}(\top)$, whose elements are (the code for) the non-dependent sets, but using an equality between its elements induced by the *intensional* equality between sets, one obtains an inconsistent extension of iTT [Jac89]. On the contrary, we will prove that any extension of iTT with a power-set as proposed in the previous section, i.e. where the equality between two elements of a power-set is induced by the provability equivalence, is consistent or at least it is not inconsistent because of the rules we proposed on the power-sets and the conditions we required.

The easiest way to prove such a result is to show first that iTT^P can be embedded in the extensional theory eTT^Ω, which is an extension of the extensional version of type theory eTT only with the power-set $\Omega \equiv \mathcal{P}(\top)$ of all the subsets of the one element set $\top$. Then we will prove that such a theory is consistent.

Thus we have the following formation and introduction rules

$$\Omega \ \mathsf{set} \qquad \Omega = \Omega \qquad \frac{U(x) \ \mathsf{set} \ [x : \top]}{\{(x : \top) \ U(x)\} \in \Omega}$$

Moreover, we require that the introduction equality condition holds, i.e. if $U(x) \leftrightarrow V(x) \ \mathsf{true} \ [x : \top]$ then there exists a proof-term $\mathsf{c}(U, V)$ such that

$$\mathsf{c}(U, V) \in \{(x : \top) \ U(x)\} =_\Omega \{(x : \top) \ V(x)\}$$

where if $x, y : \Omega$ then $x =_\Omega y$ is the abbreviation for the extensional propositional equality set $\mathsf{Eq}(\Omega, x, y)$.

Now, the condition on the existence of a proposition $a \varepsilon c \ \mathsf{set} \ [a : \top, c : \Omega]$ can be satisfied by putting, for any $c \in \Omega$,

$$a \varepsilon c \equiv (c =_\Omega \top_\top)$$

where $\top_\top \equiv \{(x : \top) \ x =_\top x\}$; here, any reference to the element a disappears in the definiens because all the elements in $\top$ are equal. Finally, we require that

$$\text{if} \ \ U(x) \ \mathsf{set} \ [x : \top] \ \ \text{then} \ \ (\{(x : \top) \ U(x)\} =_\Omega \top_\top) \leftrightarrow U(w) \ \mathsf{true} \ [w : \top]$$

Now, any power-set can be defined by putting

$$\mathcal{P}(S) \equiv S \to \Omega$$

since, for any proposition $U(x) \ \mathsf{set} \ [x : S]$, one obtains an element in $\mathcal{P}(S)$ by putting

$$\{(x : S) \ U(x)\} \equiv \lambda((x : S) \ \{(w : \top) \ U(x)\})$$

where we suppose that w does not appear free in $U(x)$, which is in fact an element in $S \to \Omega$.

Then the equality introduction condition holds provided that the propositional equality on functions is at least weakly extensional, i.e. for $f, g : A \to B$,

$$(\forall x \in A) \ (f(x) =_B g(x)) \ \to (\lambda x.f(x) =_{A \to B} \lambda x.g(x))$$

is inhabited, as it happens when the extensional version of type theory is considered.

Moreover, for any element $c \in \mathcal{P}(S)$, i.e. a function from S into Ω, and any element $a \in S$, one obtains a proposition by putting

$$a \varepsilon c \equiv (c(a) =_\Omega \top_\top)$$

which indeed satisfies the required equality condition.

Thus, any proof of $c \in \bot$ in iTT^P, i.e. any inconsistency in iTT^P, can be reconstructed in eTT^Ω. Hence, it is sufficient to show that this new theory is consistent and this will be done by defining an interpretation $\mathcal{I}$ of this theory into Zermelo-Fraenkel set theory with the axiom of choice ZFC.

The basic idea is to interpret any non-dependent set A into a set $\mathcal{I}(A)$ of ZFC and, provided that

$\mathcal{I}(A_1)$ is a set of ZFC,

$\mathcal{I}(A_2)$ is a map from $\mathcal{I}(A_1)$ into the collection of all sets of ZFC,

$\ldots$,

$\mathcal{I}(A_n)$ is a map from the disjoint union

$$\biguplus_{\alpha_1 \in \mathcal{I}(A_1), \ldots, \alpha_{n-2} \in \mathcal{I}(A_{n-2})(\langle \alpha_1, \ldots, \alpha_{n-3} \rangle)} \mathcal{I}(A_{n-1})(\langle \alpha_1, \ldots, \alpha_{n-2} \rangle)$$

into the collection of all sets of ZFC, then the dependent set

$$A(x_1, \ldots, x_n) \ \mathsf{set} \ [x_1 : A_1, \ldots, x_n : A_n(x_1, \ldots, x_{n-1})],$$

i.e. the propositional function $A : (x_1 : A_1) \ldots (x_n : A_n(x_1, \ldots, x_{n-1})) \ \mathsf{set}$, is interpreted into a map from the disjoint union

$$\biguplus_{\alpha_1 \in \mathcal{I}(A_1), \ldots, \alpha_{n-1} \in \mathcal{I}(A_{n-1})(\langle \alpha_1, \ldots, \alpha_{n-2} \rangle)} \mathcal{I}(A_n)(\langle \alpha_1, \ldots, \alpha_{n-1} \rangle)$$

into the collection of all sets of ZFC.

Since the axiom of replacement allows to avoid the use of maps into the *collection* of all sets, which can be substituted by indexed families of sets, all the interpretation can be explained within basic ZFC, but we think that the approach we use here is more perspicuous and well suited for the interpretation of a theory like eTT^{Ω} where propositional functions have to be considered.

The interpretation $\mathcal{I}(a)$ of a closed term $a \in A$, where A is a non-dependent set, will be an element of the set $\mathcal{I}(A)$ whereas the interpretation of a not-closed term

$$a(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n)\ [x_1 : A_1, \ldots, x_n : A_n(x_1, \ldots, x_{n-1})],$$

i.e. the function-element $a : (x_1 : A_1) \ldots (x_n : A_n(x_1, \ldots, x_{n-1}))\ A(x_1, \ldots, x_n)$, is a function $\mathcal{I}(a)$ which, when applied to the element

$$\alpha \in \bigcup_{\alpha_1 \in \mathcal{I}(A_1), \ldots, \alpha_{n-1} \in \mathcal{I}(A_{n-1})(\langle \alpha_1, \ldots, \alpha_{n-2} \rangle)} \mathcal{I}(A_n)(\langle \alpha_1, \ldots, \alpha_{n-1} \rangle)$$

gives the element $\mathcal{I}(a)(\alpha)$ of the set $\mathcal{I}(A)(\alpha)$.

Now, for the basic sets we put:

$$\begin{aligned} &\mathcal{I}(\bot) \equiv \emptyset \\ &\mathcal{I}(\top) \equiv \{\emptyset\} \\ &\mathcal{I}(\mathsf{Boole}) \equiv \{\emptyset, \{\emptyset\}\} \end{aligned}$$

and there is an obvious interpretation of their elements. Moreover, the sets $\Sigma(A, B)$ and $\Pi(A, B)$ (or, equivalently, the propositions $(\exists x \in A)\ B(x)$ and $(\forall x \in A)\ B(x)$) are interpreted respectively in the disjoint union and the indexed product of the interpretation of $B(x)$ indexed on the elements of the interpretation of A. The disjoint sum set $A + B$ is interpreted in the disjoint union of the interpretation of A and B and the interpretation of the extensional equality proposition $a =_A b$ is the characteristic function of the equality of the interpretation of a and b.

Finally, the interpretation of the set Ω is the set $\{\emptyset, \{\emptyset\}\}$.

Moreover, the judgement $A(x_1, \ldots, x_n)\ \mathsf{true}\ [\Gamma]$ is interpreted in $\mathcal{I}(A)(\gamma) \neq \emptyset$ for every $\gamma \in \mathcal{I}(\Gamma)$, which gives $\mathcal{I}(A) \neq \emptyset$ when A is a non-dependent set.

The interpretation of all the terms is straightforward; thus, here we only illustrate the interpretation of the elements related to the set Ω:

$$\mathcal{I}(\{(x : \top)\ U(x)\}) \equiv \begin{cases} \emptyset & \text{if } \mathcal{I}(U(*)) = \emptyset \\ \{\emptyset\} & \text{if } \mathcal{I}(U(*)) \neq \emptyset \end{cases}$$

and $\mathcal{I}(\mathsf{c}(U, V)) \equiv \emptyset$.

After these definitions, for any subset U of $\top$,

$$\mathcal{I}((\{(x : \top)\ U(x)\} =_{\Omega} \top_{\top}) \leftrightarrow U(*)) \neq \emptyset$$

by the axiom of choice and hence the equality condition is valid.

It is tedious, but straightforward, to check that all the rules of eTT^{Ω} are valid in this interpretation and hence that any proof of the judgement $a \in \bot$ within eTT^{Ω}, i.e. any form of inconsistency, would result in a proof that there is some element in $\emptyset$, that is, an inconsistency in ZFC.

7.5 iTT^{P} is classical

We are going to prove that iTT^{P} gives rise to classical logic, that is, for any proposition A the judgement $A \vee \neg A\ \mathsf{true}$ holds. Even if iTT^{P} is *not* a topos, the proof that we show here is obtained by adapting to our framework an analogous result stating that any topos satisfying the axiom of choice is boolean. Among the various proofs of this result (see for instance [LS86] and [Bel88]), which goes back to Diaconescu's work showing that one obtains ZF by adding the axiom of choice to IZF (see [Dia75]), we choose to translate the proof of Bell [Bel88], because it is very well suited to work in iTT^{P} since it is almost completely developed within local set theory instead of topos theory, except for the use of a choice rule.

In iTT^P the result is a consequence of the strong elimination rule for disjoint union which allows to prove the so called *intuitionistic axiom of choice*, i.e.

$$((\forall x \in A)(\exists y \in B)\ C(x,y)) \to ((\exists f \in A \to B)(\forall x \in A)\ C(x, f(x)))\ \mathsf{true}$$

Let us recall the proof [Mar84]. Assume that $h \in (\forall x \in A)(\exists y \in B)\ C(x,y)$ and that $x \in A$. Then $h(x) \in (\exists y \in B)\ C(x,y)$. Let $\mathsf{fst}(-)$ and $\mathsf{snd}(-)$ be the first and the second projection respectively; then the elimination rule for the set of the disjoint union allows to prove that

$$\mathsf{fst}(h(x)) \in B$$

and

$$\mathsf{snd}(h(x)) \in C(x, \mathsf{fst}(h(x)))$$

Hence, by putting $f \equiv \lambda x.\mathsf{fst}(h(x))$ we obtain both

$$f \in A \to B$$

and

$$\mathsf{snd}(h(x)) \in C(x, f(x))$$

since, by β-equality, $f(x) \equiv (\lambda x.\mathsf{fst}(h(x)))(x) = \mathsf{fst}(h(x))$. Finally, we conclude by using the *true*-introduction rule.

Now, in the sequel, we will first show the structure of the proof skipping the formalization details and then we will formalize it inside Martin-Löf's type theory.

First note that for any $U, V \in \mathcal{P}(\top)$, if $\mathsf{decode}(U) \vee \mathsf{decode}(V)$ holds then there exists an element $x \in \mathsf{Boole}$ such that

$$(x =_{\mathsf{Boole}} \mathsf{true} \to \mathsf{decode}(U)) \wedge (x =_{\mathsf{Boole}} \mathsf{false} \to \mathsf{decode}(V))$$

because $\neg(\mathsf{true} =_{\mathsf{Boole}} \mathsf{false})$ is provable in Martin-Löf's type theory with one universe. Then, by the axiom of choice, there exists a function f such that, for any $U, V \in \mathcal{P}(\top)$ such that $\mathsf{decode}(U) \vee \mathsf{decode}(V)$ holds,

$$(f(\langle U, V\rangle) =_{\mathsf{Boole}} \mathsf{true} \to \mathsf{decode}(U)) \wedge (f(\langle U, V\rangle) =_{\mathsf{Boole}} \mathsf{false} \to \mathsf{decode}(V))$$

Now, let A be any proposition. Then $\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle$ and $\langle \mathsf{code}(\top), \mathsf{code}(A)\rangle$ are two couples such that

$$\mathsf{decode}(\mathsf{code}(A)) \ \vee\ \mathsf{decode}(\mathsf{code}(\top))$$

and

$$\mathsf{decode}(\mathsf{code}(\top)) \ \vee\ \mathsf{decode}(\mathsf{code}(A))$$

hold because $\mathsf{decode}(\mathsf{code}(\top)) \leftrightarrow \top$. Then, with a bit of intuitionistic logic, we can obtain both

$$f(\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle) =_{\mathsf{Boole}} \mathsf{true} \to \mathsf{decode}(\mathsf{code}(A))$$

and

$$f(\langle \mathsf{code}(\top), \mathsf{code}(A)\rangle) =_{\mathsf{Boole}} \mathsf{false} \to \mathsf{decode}(\mathsf{code}(A))$$

and hence

$$f(\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle) =_{\mathsf{Boole}} \mathsf{true} \to A$$

and

$$f(\langle \mathsf{code}(\top), \mathsf{code}(A)\rangle) =_{\mathsf{Boole}} \mathsf{false} \to A$$

because $\mathsf{decode}(\mathsf{code}(A)) \leftrightarrow A$.

But we know that the set Boole is decidable (see [NPS90]) and hence

$$f(\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle) =_{\mathsf{Boole}} \mathsf{true} \vee f(\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle) =_{\mathsf{Boole}} \mathsf{false}$$

holds. Thus we can argue by $\vee$-elimination as follows. If

$$f(\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle) =_{\mathsf{Boole}} \mathsf{true}$$

then $f(\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle) =_{\mathsf{Boole}} \mathsf{true} \to A$ yields A and hence

$$A \vee f(\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle) =_{\mathsf{Boole}} \mathsf{false}$$

holds. On the other hand $f(\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle) =_{\mathsf{Boole}} \mathsf{false}$ yields directly

$$A \vee f(\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle) =_{\mathsf{Boole}} \mathsf{false}$$

Similarly we can obtain

$$A \vee f(\langle \mathsf{code}(\top), \mathsf{code}(A)\rangle) =_{\mathsf{Boole}} \mathsf{true}$$

Hence, by distributivity, we have

$$A \ \vee \ (f(\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle) =_{\mathsf{Boole}} \mathsf{false} \ \wedge \ f(\langle \mathsf{code}(\top), \mathsf{code}(A)\rangle) =_{\mathsf{Boole}} \mathsf{true})$$

and we can argue by $\vee$-elimination to prove that $A \ \vee \ \neg A$ holds.

In fact, assuming that A holds yields directly that $A \ \vee \ \neg A$ holds.

On the other hand, let us suppose that

$$f(\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle) =_{\mathsf{Boole}} \mathsf{false}$$

and

$$f(\langle \mathsf{code}(\top), \mathsf{code}(A)\rangle) =_{\mathsf{Boole}} \mathsf{true}$$

and assume that A is true. Then $A \leftrightarrow \top$ holds, and hence $\mathsf{code}(A) =_{\mathcal{P}(\top)} \mathsf{code}(\top)$ by equality introduction. Thus

$$\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle =_{\mathcal{P}(\top)\times\mathcal{P}(\top)} \langle \mathsf{code}(\top), \mathsf{code}(A)\rangle$$

and hence

$$\mathsf{false} =_{\mathsf{Boole}} f(\langle \mathsf{code}(A), \mathsf{code}(\top)\rangle) =_{\mathsf{Boole}} f(\langle \mathsf{code}(\top), \mathsf{code}(A)\rangle) =_{\mathsf{Boole}} \mathsf{true}$$

So we are arrived to a contradiction starting from the assumption that A holds, and thus $\neg A$ holds which gives also in this case that $A \ \vee \ \neg A$ holds.

Thus we proved that classical logic is yielded by our two conditions. Of course we did not furnish a proof element for $A \vee \neg A$ since we only required that for any $a \in S$ and $U \subseteq S$, $a\varepsilon\{x \in S |\ U(x)\} \leftrightarrow U(a)$ holds but we could not furnish a proof element for this judgement, that is we destroyed the correspondence between the judgements A true and the fact that there exists a proof-term a such that $a \in A$ (see [Val98]).

Let us now show how the previous argument can be completely formalized inside Martin-Löf's type theory.

Since in the following we will mainly use the power-set $\mathcal{P}(\top)$, we introduce some abbreviations besides of $\Omega \equiv \mathcal{P}(\top)$ and $\top_\top \equiv \{(w : \top)\ w =_\top w\}$ already used in section 7.4; let us suppose that U is any proposition and $w : \top$ is a variable which does not appear free in U, then we put $[U] \equiv \{(w : \top)\ U\}$ and, supposing $p \in \Omega$, we put $\overline{p} \equiv *\varepsilon p$. Moreover, following a standard practice, supposing A is a proposition, sometimes we will simply write A to assert the judgement A true.

It is convenient to state here all the properties of the intensional equality proposition Id that we need in the following. First, we recall two well known results: Id is an equivalence relation, and if A and B are sets and $a =_A c$ and $f =_{A\to B} g$ then $f(a) =_B g(c)$ (for a proof see [NPS90], page 64).

Moreover, the following properties of Id are specific of the new set Ω. They are similar to the properties that the set Id enjoys when it is used on elements of the set U, i.e. the universe of the small sets, which we will not use at all. In fact, Ω resembles this set, but it also differs because of the considered equality and because a code for each set is present in Ω whereas only the codes for the small sets can be found in U.

Lemma 7.5.1 *If $p =_\Omega q$ then $\overline{p} \leftrightarrow \overline{q}$.*

Proof. Let $x \in \Omega$; then $\overline{x} \leftrightarrow \overline{x}$ and hence $\overline{p} \leftrightarrow \overline{q}$ is a consequence of $p =_\Omega q$ by Id-elimination. □

Lemma 7.5.2 $\neg(\mathsf{true} =_{\mathsf{Boole}} \mathsf{false})$.

Proof. Let $x \in \mathsf{Boole}$; then $\mathsf{if}\ x\ \mathsf{then}\ [\top]\ \mathsf{else}\ [\bot] \in \Omega$. Now, suppose that $\mathsf{true} =_{\mathsf{Boole}} \mathsf{false}$, then $\mathsf{if\ true\ then}\ [\top]\ \mathsf{else}\ [\bot] =_\Omega \mathsf{if\ false\ then}\ [\top]\ \mathsf{else}\ [\bot]$ which yields $[\top] =_\Omega [\bot]$ by boole-equality and transitivity. Thus, by the previous lemma $\overline{[\top]} \leftrightarrow \overline{[\bot]}$, but $\overline{[\top]} \leftrightarrow \top$ and $\overline{[\bot]} \leftrightarrow \bot$ by the equality condition; hence $\bot\ \mathsf{true}$ and thus, by discharging the assumption $\mathsf{true} =_{\mathsf{Boole}} \mathsf{false}$, we obtain the result. □

Now, we will start the proof of the main result of this section. The trick to internalize the proof in [Bel88] within iTT^P is stated in the following lemma.

Lemma 7.5.3 *For any proposition A, if $A\ \mathsf{true}$ then*

$$\mathsf{c}((w : \top)\ A, (w : \top)\ w =_\top w) \in [A] =_\Omega \top_\top$$

and hence $[A] =_\Omega \top_\top\ \mathsf{true}$; moreover, if $[A] =_\Omega \top_\top\ \mathsf{true}$ then $A\ \mathsf{true}$.

Proof. If $A\ \mathsf{true}$ then $A \leftrightarrow (w =_\top w)\ \mathsf{true}\ [w : \top]$; hence, by the equality introduction condition, $\mathsf{c}((w : \top)\ A, (w : \top)\ w =_\top w) \in [A] =_\Omega \top_\top$, and thus $[A] =_\Omega \top_\top\ \mathsf{true}$ by *true-introduction*; on the other hand, if $[A] =_\Omega \top_\top\ \mathsf{true}$ then $\overline{[A]} \leftrightarrow \overline{\top_\top}$ by lemma 7.5.1, but $\overline{[A]} \leftrightarrow A$ and $* =_\top * \leftrightarrow \overline{\top_\top}$ by the equality condition, and hence $A\ \mathsf{true}$ since $* =_\top *\ \mathsf{true}$. □

After this lemma, for any proposition A it is possible to obtain a logically equivalent proposition, i.e. $[A] =_\Omega \top_\top$, such that, if $A\ \mathsf{true}$, the proof element $\mathsf{c}((w : \top)\ A, (w : \top)\ w =_\top w)$ of $[A] =_\Omega \top_\top$ has no memory of the proof element which testifies the truth of A. We will see that this property is crucial to get the main result. We will use the above lemma immediately in the next one where, instead of the proposition $\overline{\mathsf{fst}(w)} \vee \overline{\mathsf{snd}(w)}\ \mathsf{set}\ [w : \Omega \times \Omega]$, we write $[\overline{\mathsf{fst}(w)} \vee \overline{\mathsf{snd}(w)}] =_\Omega \top_\top\ \mathsf{set}\ [w : \Omega \times \Omega]$ in order to avoid that the proof-term in the main statement depends on the truth of the first or of the second disjunct.

Proposition 7.5.4 *In iTT^P the following proposition*

$$\begin{array}{r}(\forall z \in \Sigma(\Omega \times \Omega, (w)\ [\overline{\mathsf{fst}(w)} \vee \overline{\mathsf{snd}(w)}] =_\Omega \top_\top)) \\ (\exists x \in \mathsf{Boole})\ (x =_{\mathsf{Boole}} \mathsf{true} \rightarrow \overline{\mathsf{fst}(\mathsf{fst}(z))}) \wedge \\ (x =_{\mathsf{Boole}} \mathsf{false} \rightarrow \overline{\mathsf{snd}(\mathsf{fst}(z))})\end{array}$$

is true.

Proof. Suppose $z \in \Sigma(\Omega \times \Omega, (w)\ [\overline{\mathsf{fst}(w)} \vee \overline{\mathsf{snd}(w)}] =_\Omega \top_\top)$ then $\mathsf{fst}(z) \in \Omega \times \Omega$ and $\mathsf{snd}(z)$ is a proof of $[\overline{\mathsf{fst}(\mathsf{fst}(z))} \vee \overline{\mathsf{snd}(\mathsf{fst}(z))}] =_\Omega \top_\top$. Thus, by lemma 7.5.3, $\overline{\mathsf{fst}(\mathsf{fst}(z))} \vee \overline{\mathsf{snd}(\mathsf{fst}(z))}$. Now, the result can be proved by $\vee$-elimination. In fact, if

$$\overline{\mathsf{fst}(\mathsf{fst}(z))}\ \mathsf{true}$$

then

$$\mathsf{true} =_{\mathsf{Boole}} \mathsf{true} \rightarrow \overline{\mathsf{fst}(\mathsf{fst}(z))}$$

Moreover, $\neg(\mathsf{true} =_{\mathsf{Boole}} \mathsf{false})$ by lemma 7.5.2 and hence

$$\mathsf{true} =_{\mathsf{Boole}} \mathsf{false} \rightarrow \overline{\mathsf{snd}(\mathsf{fst}(z))}$$

Thus we obtain that

$$(\exists x \in \mathsf{Boole})\ (x =_{\mathsf{Boole}} \mathsf{true} \rightarrow \overline{\mathsf{fst}(\mathsf{fst}(z))}) \wedge (x =_{\mathsf{Boole}} \mathsf{false} \rightarrow \overline{\mathsf{snd}(\mathsf{fst}(z))})$$

On the other hand, by a similar proof we reach the same conclusion starting from the assumption $\overline{\mathsf{snd}(\mathsf{fst}(z))}\ \mathsf{true}$. □

Thus, we can use the intuitionistic axiom of choice to obtain:

Proposition 7.5.5 *In* iTT^P *the following proposition*

$$\begin{array}{l}(\exists f \in \Sigma(\Omega \times \Omega, (w)\ [\overline{\mathsf{fst}(w)}\ \vee\ \overline{\mathsf{snd}(w)}] =_\Omega \top_\top) \to \mathsf{Boole})\\ \quad(\forall z \in \Sigma(\Omega \times \Omega, (w)\ [\overline{\mathsf{fst}(w)}\ \vee\ \overline{\mathsf{snd}(w)}] =_\Omega \top_\top))\\ \quad\quad(f(z) =_{\mathsf{Boole}} \mathsf{true}\ \to\ \overline{\mathsf{fst}(\mathsf{fst}(z))}\ \wedge\ (f(z) =_{\mathsf{Boole}} \mathsf{false}\ \to\ \overline{\mathsf{snd}(\mathsf{fst}(z))})\end{array}$$

is true.

Now, suppose that A is any proposition; then

$$\langle\langle [A], \top_\top\rangle, \mathsf{k}([A], \top_\top)\rangle \in \Sigma(\Omega \times \Omega, (w)\ [\overline{\mathsf{fst}(w)}\ \vee\ \overline{\mathsf{snd}(w)}] =_\Omega \top_\top)$$

where $\mathsf{k}(x, y)$ is short for $\mathsf{c}((w : \top)\ \overline{\mathsf{fst}(\langle x, y\rangle)}\ \vee\ \overline{\mathsf{snd}(\langle x, y\rangle)}, (w : \top)\ w =_\top w)$.

In fact, $\langle [A], \top_\top\rangle \in \Omega \times \Omega$ and $\overline{\top_\top}$ true, hence $\overline{\mathsf{fst}(\langle [A], \top_\top\rangle)}\ \vee\ \overline{\mathsf{snd}(\langle [A], \top_\top\rangle)}$; thus the result follows by lemma 7.5.3.

Now, let f be the choice function in the proposition 7.5.5; then

$$f(\langle\langle [A], \top_\top\rangle, \mathsf{k}([A], \top_\top)\rangle) =_{\mathsf{Boole}} \mathsf{true}\ \to\ \overline{[A]}$$

But

$$(f(\langle\langle [A], \top_\top\rangle, \mathsf{k}([A], \top_\top)\rangle) =_{\mathsf{Boole}} \mathsf{true}) \vee (f(\langle\langle [A], \top_\top\rangle, \mathsf{k}([A], \top_\top)\rangle) =_{\mathsf{Boole}} \mathsf{false})$$

since the set Boole is decidable (for a proof see [NPS90], page 177), and hence, by $\vee$-elimination and a little of intuitionistic logic, one obtains that

$$(1)\quad \overline{[A]}\ \vee\ (f(\langle\langle [A], \top_\top\rangle, \mathsf{k}([A], \top_\top)\rangle) =_{\mathsf{Boole}} \mathsf{false})$$

Analogously one can prove that

$$(2)\quad \overline{[A]}\ \vee\ (f(\langle\langle \top_\top, [A]\rangle, \mathsf{k}(\top_\top, [A])\rangle) =_{\mathsf{Boole}} \mathsf{true})$$

Thus, by using distributivity on the conjunction of (1) and (2), one finally obtains

Proposition 7.5.6 *For any proposition* A *in* iTT^P *the following proposition*

$$\begin{array}{l}(\exists f \in \Sigma(\Omega \times \Omega, (w)\ [\overline{\mathsf{fst}(w)}\ \vee\ \overline{\mathsf{snd}(w)}] =_\Omega \top_\top) \to \mathsf{Boole})\\ \quad\overline{[A]} \vee ((f(\langle\langle [A], \top_\top\rangle, \mathsf{k}([A], \top_\top)\rangle) =_{\mathsf{Boole}} \mathsf{false}) \wedge\\ \quad\quad(f(\langle\langle \top_\top, [A]\rangle, \mathsf{k}(\top_\top, [A])\rangle) =_{\mathsf{Boole}} \mathsf{true}))\end{array}$$

is true.

Now, let us assume $\overline{[A]}$ true; then $[A] =_\Omega \top_\top$ true by lemma 7.5.3 and hence

$$\langle\langle [A], \top_\top\rangle, \mathsf{k}([A], \top_\top)\rangle =_{\Sigma(\Omega\times\Omega,\ldots)} \langle\langle \top_\top, \top_\top\rangle, \mathsf{k}(\top_\top, \top_\top)\rangle$$

since

$$\lambda x.\langle\langle x, \top_\top\rangle, \mathsf{k}(x, \top_\top)\rangle \in \Omega \to \Sigma(\Omega \times \Omega, (w)\ [\overline{\mathsf{fst}(w)}\ \vee\ \overline{\mathsf{snd}(w)}] =_\Omega \top_\top)$$

Thus

$$f(\langle\langle [A], \top_\top\rangle, \mathsf{k}([A], \top_\top)\rangle) =_{\mathsf{Boole}} f(\langle\langle \top_\top, \top_\top\rangle, \mathsf{k}(\top_\top, \top_\top)\rangle)$$

where f is the function whose existence is stated by the proposition 7.5.6.

With the same assumption, also

$$f(\langle\langle \top_\top, [A]\rangle, \mathsf{k}(\top_\top, [A])\rangle) =_{\mathsf{Boole}} f(\langle\langle \top_\top, \top_\top\rangle, \mathsf{k}(\top_\top, \top_\top)\rangle)$$

can be proved in a similar way; hence, by transitivity of the equality proposition,

$$f(\langle\langle [A], \top_\top\rangle, \mathsf{k}([A], \top_\top)\rangle) =_{\mathsf{Boole}} f(\langle\langle \top_\top, [A]\rangle, \mathsf{k}(\top_\top, [A])\rangle)$$

It is worth noting that this result depends mainly on lemma 7.5.3, and hence on the equality introduction condition whose premise is a "true judgement". Indeed, $\lambda x.\langle\langle x, \top_\top\rangle, \mathsf{k}(x, \top_\top)\rangle$ and

$\lambda x.\langle\langle \top_\top, x\rangle, \mathsf{k}(\top_\top, x)\rangle$ yield equal results when applied to $\top_\top$ since they do not depend on the proof-terms used to derive the two judgements

$$(\overline{x} \vee \overline{\top_\top}) \leftrightarrow (w =_\top w) \ \mathsf{true} \ [x : \Omega, w : \top]$$

and

$$(\overline{\top_\top} \vee \overline{x}) \leftrightarrow (w =_\top w) \ \mathsf{true} \ [x : \Omega, w : \top]$$

In the case we admit dependency on the proof-terms in the equality introduction condition we can redo the whole proof if we assume that uniqueness of equality proofs (see the rules in [HS95] or [Hof95]) holds and we replace $\overline{a}$ with $a =_\Omega \top_\top$, where $a \in \Omega$, everywhere in the proof in order to get an actual proof-term at this point.

Now, by assuming both $\overline{[A]}$ true and

$$(f(\langle\langle [A], \top_\top\rangle, \mathsf{k}([A], \top_\top)\rangle) =_{\mathsf{Boole}} \mathsf{false}) \wedge (f(\langle\langle \top_\top, [A]\rangle, \mathsf{k}(\top_\top, [A])\rangle) =_{\mathsf{Boole}} \mathsf{true})$$

one can conclude $\mathsf{true} =_{\mathsf{Boole}} \mathsf{false}$.

On the other hand, we know that $\neg(\mathsf{true} =_{\mathsf{Boole}} \mathsf{false})$ holds by lemma 7.5.2. Hence $\bot$ follows and so we obtain that the judgement $\neg\overline{[A]}$ true holds by discharging the assumption $\overline{[A]}$ true. Then, by using proposition 7.5.6 and a little of intuitionistic logic, we can conclude $(\overline{[A]} \vee \neg\overline{[A]})$ true which, by the equality condition, yields $(A \vee \neg A)$ true. Thus, if we could give suitable rules for the power-sets that allow our conditions to hold and follow the usual meaning for the judgement C true, i.e. C true holds if and only if there exists a proof element for the proposition C, then we would have a proof element for the proposition $A \vee \neg A$, which is expected to fail.

7.6 Some remarks on the proof

To help the reader who knows the proof in [Bel88], it may be useful to explain the differences between the original proof and that presented in the previous section. Our proof is not the plain application of Bell's result to iTT^P since iTT^P is not a topos. It is possible to build a topos out of the extensional theory eTT^P obtained by adding a power-set constructor to eTT, if one adds to it also the rule of η-equality for power-sets, like in the end of section 7.3. However, we showed that it is not necessary to be within a topos to reconstruct Diaconescu's result and that a weaker theory is sufficient.

This fact suggests that it is not possible to extend Martin-Löf's set theory, where sets and propositions are identified and proof-elements can be provided for any provable proposition, to an intuitionistic theory of sets fully equipped with power-sets satisfying the conditions discussed in section 7.3, provided that we want to preserve the constructive meaning of the connectives. However, observe that the requirement of uniformity in the extensional equality condition is crucial to carry on our proof. Therefore, it seems that there is still some hope to get power-sets in constructive type theory by dropping uniformity. But, it is worth recalling that an analogous proof of excluded middle can be performed also without uniformity if the uniqueness of equality proofs holds. Thus, no constructive power-set can be added if type theory is endowed with pattern matching, which is usually used in most of its actual implementations.

7.7 Other dangerous set constructions

In the previous sections we proved that by adding an extensional set constructor to Martin-Löf's type theory, which satisfies some vary natural conditions, classical logic is obtained. In the next sections we will show that the problem is not due to the power set constructor but mainly to the required extesionality.

The need for the addition of extensional set constructors to Martin-Löf's type theory appears immediately as soon as one tries to develop within such a framework a piece of actual mathematics, as we did with constructive topology (see the introductory sections in [CSSV]). In fact, if one works on constructive topology, and hence within the set theoretical framework of Martin-Löf's constructive type theory, it is very likely that he is going to wonder on the reason way one wants to

develop topology within a framework which is so difficult to deal with. In particular, many useful set-constructors are not present there, while it is clear that they would be very useful in order to work with topological concepts. For instance, one would like to be able to work with power-sets, in order to quantify on the elements of the collection of the subsets of a set, or with quotient-sets, in order to be able to construct in a straightforward way some well known sets, or, at least, with the collection of the finite subsets of a set, in order, for instance, to be able to express easily statements like Tychonoff's theorem on the compactness of the product of compact spaces. Of course, the main point is to be able to deal with such constructions and still work in a set-theory with such a clear semantics as Martin-Löf's type theory has. The bad news is that there is no possibility to be sure that this is possible at all, as we will see in this paper.

7.7.1 The collection of the finite subsets

It is interesting to observe that all of the proof in the previous section can be redone also if we assume a set-constructor which seems to be much more safe than the power-set constructor, namely the set $\mathcal{P}_\omega(S)$ of the finite subsets of the set S.

In order to define $\mathcal{P}_\omega(S)$ we have first to state what a finite subset is. The first idea is probably to identify a finite subset of S with a list of elements of S, but if extensionality is required we have to force two lists to be equal if they are formed with the same elements, that is, we have to force a quotient over the set of the lists on S and this quotient is going to be based on an equivalence relation which will be defined in terms of the equality over S which can be non-decidable. And we will see that also quotient sets over non decidable equivalence relations are not safe set-constructions.

The good news is that we have a possible way out. In fact, we can find a suitable condition which allows to state which ones are finite among the standard subsets. To this aim, let $\mathsf{N}(k)$ prop $[k : \mathsf{N}]$ be the family of sets defined over the natural numbers by using the following inductive definition

$$\begin{cases} \mathsf{N}(0) & = & \bot \\ \mathsf{N}(k+1) & = & \mathsf{S}(\mathsf{N}(k)) \end{cases}$$

where the type $\mathsf{S}(A)$ is the set-constructor which, given any set A, allows to obtain a new set with one element more than A. Its formation and introduction rules are the following:

$$\frac{A\ \mathsf{set}}{\mathsf{S}(A)\ \mathsf{set}}$$

$$0_{\mathsf{S}(A)} \in \mathsf{S}(A) \qquad \frac{a \in A}{\mathsf{s}_{\mathsf{S}(A)}(a) \in \mathsf{S}(A)}$$

So, for any $k \in \mathsf{N}$, the set $\mathsf{N}(k)$ contains exactly k elements.

Now, given any subset U of S, that is any propositional function over S, we can put

$$\mathsf{Fin}(U) \equiv (\exists k \in \mathsf{N})(\exists f \in \mathsf{N}(k) \to S)\ U \subseteq \mathsf{Im}(f)$$

where $\mathsf{Im}(f) \equiv \{x \in S |\ (\exists n \in \mathsf{N}(k))\ x =_S f[n]\}$. The previous definition states that a subset U is finite whenever it is a subsets of a surely finite subset, namely the image of a finite set.

This is a good definition of "finite subset" because it follows our intuition about what a finite subset should be and meanwhile it does not force us to know how many elements the subset should contains but it only require that there is a finite upper bound to the number of elements that it can contain. This fact allows to prove some expected result on finite subsets which were not provable with the approach in [SV98]. For instance, here not only the union of two finite subsets is finite but also the intersection of a finite subset with any other subset is finite and this fact would have been difficult to prove if the notion of finite subset would have required to know the number of its elements.

A consequence of the previous definition is that we can state the introduction rule for the set $\mathcal{P}_{\mathsf{fin}}(S)$ by putting

$$\frac{U(x)\ \mathsf{prop}\ [x : S] \quad \mathsf{Fin}(U)\ \mathsf{true}}{\{x \in S |\ U(x)\} \in \mathcal{P}_{\mathsf{fin}}(S)}$$

But then we obtain that all of the subsets of the one-element set are finite and hence $\mathcal{P}_{\mathsf{fin}}(\top)$ and $\mathcal{P}(\top)$ are (almost) the same set and hence all of the proof in the previous section can be redone using the set $\mathcal{P}_{\mathsf{fin}}(\top)$.

7.7.2 The quotient set constructor

We observed above that it would have been possible to define the set of the finite subsets of a given set S also considering the set of the list over S and then forcing a quotient over it in order to obtain the wished extensionality. But it is possible to prove that also a quotient set constructor is not safe from a constructive point of view.

Indeed, the quotient set-constructor can be defined by using the following formation and introduction rules

$$\frac{A \text{ set} \quad R(x,y) \text{ prop } [x,y:A] \quad \mathsf{EqRel}(R) \text{ true}}{A_R \text{ set}}$$

$$\frac{a \in A \quad R(x,y) \text{ prop } [x,y:A] \quad \mathsf{EqRel}(R) \text{ true}}{[a]_R \in A_R}$$

where $\mathsf{EqRel}(R)$ is any proposition which formalizes the standard conditions requiring that R is an equivalence relation.

Now, in order to obtain a quotient set, we should require a suitable equality rule, but it is possible to show that this is not possible if we want to avoid to obtain classical logic and still obtain a set which is a real quotient set. In fact, if we require that, whatever equality rule one can use, the following condition is satisfied, for any $a, b \in A$,

$$R(a,b) \text{ true if and only if } [a]_R =_{A_R} [b]_R \text{ true}$$

then we can construct a proof of $A \vee \neg A$, for any small set A, by arguing as in section 7.5 but using, instead that $\mathcal{P}(\top)$, the quotient set V obtained by using first universe U_0, which contains the codes of all the small sets, and the equivalence relation of equi-provability between small types (for a detailed proof see [Mai99]).

7.7.3 The two-subset set

In the previous sections we proved that by adding extensional set-constructors to Martin-Löf's type theory one can obtain classical logic. The obvious question is: where is the problem? Here we will prove that even a very weak form of extensionality seems not to fit well with constructive type theory. Indeed, it is possible to obtain classical logic even considering like a set just the collection of *two* subsets, if an exstensional equality is required on the elements of this set. In fact, let us suppose to add to Martin-Löf's type theory the following formation rule

$$\frac{U(x) \text{ prop } [x:S] \qquad V(x) \text{ prop } [x:S]}{\{U,V\} \text{ set}}$$

and then suitable introduction and elimination rules in such a way that the following very natural conditions are satisfied

1. (pair axiom) If $W \subseteq S$ then

$$W \in \{U,V\} \text{ if and only if } (W =^S U) \ \vee \ (W =^S V) \text{ true}$$

2. (extensionality) If $W_1, W_2 \in \{U,V\}$ then

$$\text{if } W_1 =^S W_2 \text{ true then } W_1 =_{\{U,V\}} W_2 \text{ true}$$

Then, we can formalize within this extended theory a proof by J. Bell in [Bel88] which allows to prove $A \vee \neg A$ for any proposition A. In fact, let

$$V_0 \equiv \{x \in \mathsf{Boole} |\ (x =_{\mathsf{Boole}} \mathsf{false}) \ \vee \ A\}$$
$$V_1 \equiv \{x \in \mathsf{Boole} |\ (x =_{\mathsf{Boole}} \mathsf{true}) \ \vee \ A\}$$

Then we can prove that

$$(\forall W \in \{V_0, V_1\})(\exists y \in \mathsf{Boole})\ y \varepsilon W$$

In fact, by the pair axiom, $W \in \{V_0, V_1\}$ yields $(W =^S V_0) \vee (W =^S V_1)$ and hence we can argue by $\vee$-elimination. Now, supposing $W =^S V_0$, the fact that $\mathsf{false} \varepsilon V_0$ yields that $\mathsf{false} \varepsilon W$ and then $(\exists y \in \mathsf{Boole})\ y \varepsilon W$ and, similarly, supposing $W =^S V_1$, the fact that $\mathsf{true} \varepsilon V_1$ yields that $\mathsf{true} \varepsilon W$ and then also in this case we obtain $(\exists y \in \mathsf{Boole})\ y \varepsilon W$.

Then, by the axiom of choice, we obtain

$$(\exists f \in \{V_0, V_1\} \to \mathsf{Boole})\ (\forall W \in \{V_0, V_1\})\ f(W) \varepsilon W$$

and hence

$$(f(V_0) \varepsilon V_0) \wedge (f(V_1) \varepsilon V_1)$$

Note now that $f(V_0) \varepsilon V_0$ holds if and only if $(f(V_0) =_{\mathsf{Boole}} \mathsf{false}) \vee A$ and $f(V_1) \varepsilon V_1$ holds if and only if $(f(V_1) =_{\mathsf{Boole}} \mathsf{true}) \vee A$; hence we using a bit of intuitionistic logic we obtain that

$$(f(V_0) =_{\mathsf{Boole}} \mathsf{false} \wedge f(V_1) =_{\mathsf{Boole}} \mathsf{true}) \vee A$$

and thus we can argue by $\vee$-elimination. If A holds then we obtain directly that $A \vee \neg A$ holds. On the other hand, if we assume that A holds, then $V_0 =^S V_1$, and hence, by extensionality, we obtain that $V_0 =_{\{V_0, V_1\}} V_1$. Thus, by one of the property of the intensional equality proposition, we obtain that $f(V_0) =_{\mathsf{Boole}} f(V_1)$ which together with $(f(V_0) =_{\mathsf{Boole}} \mathsf{false}) \wedge (f(V_1) =_{\mathsf{Boole}} \mathsf{true})$ gives $\mathsf{false} =_{\mathsf{Boole}} \mathsf{true}$ that leads to a contradiction when we work within a theory with the universe of the small types. Thus we can conclude that the assumption that A holds lead to a contradiction, that is $\neg A$ holds and hence also in this case $A \vee \neg A$ holds by $\vee$-introduction.

Appendix A

Expressions theory

A.1 The Expressions with arity

A.1.1 Introduction

Martin Löf's type theory [Mar84, NPS90] has been exposed by giving, besides the conceptual motivations, also a set of formal rules where types and their elements are denoted by *expressions* build up from variables and constants by means of application (notation: $b(a)$) and abstraction (notation: $((x)\ b)$). Moreover, the operational semantics of type theory is given by expression manipulations.

Since, in order to verify the correctness of the application of a rule we need, almost always, both to recognize definitional equality between textual different expressions (that is, different denotations of the same type) and to reduce expressions into equivalent ones (e.g $a \Rightarrow ((x)\ a)(x)$), we need a formal theory for expressions which deals with these problems at a level underlying type theory. Clearly pure lambda-calculus cannot be considered because of the undecidability of definitional equality due to the possibility of self-application [Chu36]. Historically, to solve this problem, typed lambda-calculus [Bar84, CF74] has been introduced.

A similar approach has been followed by Martin-Löf [Mar83, NPS90] where an arity is associated to any expression in order to specify its functionality. The main difference with respect to standard typed lambda-calculus is in the use of abbreviating definitions which extend the linguistic expressiveness of expressions instead of lambda-abstraction. Definitions seem necessary to introduce derived rules in type theory and their power is manifested by the ability to emulate abstraction.

Our approach is strongly related to Martin Löf's one, but we give a different treatment of definitions both to make them suitable for a mechanical treatment and to be able to analyze the conditions under which it is possible to develop a provable correct parsing algorithm for expressions. As a consequence other changes were needed and, for this reason, Martin-Löf has no responsibility for the following.

The chapter is organized as follows. In section A.2 you can find the definition of arity, the characterization of abbreviating definitions, the definition of the rules to form expressions with arity and the rules to prove that two expressions are definitionally equal.

In section A.3 some properties of the expression system previously defined are proved. In this system is easy to deal with both normal form and decidability of expressions: the notion of normal form is introduced very naturally and an algorithm to reduce any expression into an equivalent one in normal form is easy to develop and to prove correct.

In section A.4 we introduce new rules to form expressions and we prove that they are sufficient to form all of the expressions without introducing new ones. The new rules are introduced to meet the requirements to obtain decidability of the predicate *'to be an expression'*.

Then, in section A.5, abstraction is introduced as a schema of abbreviating definitions and the concepts of normal form and its principal properties are given.

Finally the relationships between the expression theory and the standard typed lambda-calculus with α, β, η and ξ reductions is dealt with in section A.6.

A.2 Basic definitions

The formal definition of arity is the following.

Definition A.2.1 (Arity) *An arity α is defined by induction as follows:*

1. *0 is an arity*
2. *If α and β are two arities then $(\alpha)\beta$ is an arity.*

0 will be the arity of expressions which cannot be applied, that is the *saturated* expressions. We shall subsequently usually omit the rightmost 0 of an arity since this will not lead to ambiguity.

Arities are naturally ordered by the usual ordering on construction complexity and two arities are equal if and only if they are textually identical.

In the following we assume to have a set (possibly empty) of primitive constants with arity $c - \alpha$ and, for any arity α, an enumerable set of variables $x_1 - \alpha$, $x_2 - \alpha$, ... Since we want that any variable and any constant is recognizable with its unique arity we assume the decidability of the two predicates $x - \alpha$ var and $c - \alpha$ const whose reading is "x is a variable of arity α" and "c is a primitive constant of arity α" respectively. Moreover we assume the ability to recognize the occurrences of a variable and of a primitive constant within a string.

We shall use the notation $d\{x_1, \ldots, x_n\}$ to stress that $x_1, \ldots, x_n$ are all the variables occurring within the string d.

A.2.1 Abbreviating definitions

The notion of abbreviating definition plays a central role in what follows. An abbreviating definition is a binary relation on strings:

$$\mathsf{definiendum} =_{def} \mathsf{definiens}$$

where the first element is called *definiendum* (what we are defining on) and the second one is called *definiens* (what we assume to already have). We can think of the definiendum as a new denotation for the object denoted by the definiens. The general form of a definiendum is

$$d\{x_1, \ldots, x_n\}(y_1) \ldots (y_m) \qquad n, m \geq 0$$

where distinguishable, and declared, occurrences of the variables $x_1, \ldots, x_n$ (the *name parameters*) of arity $\alpha_1, \ldots, \alpha_n$ may appear inside the string d (the *parametric name*) and $y_1, \ldots, y_m$ (the *arguments*) are variables of arity $\beta_1, \ldots, \beta_m$ respectively.

We shall call definition skeleton the result of the substitution, within a parametric name, of the name-parameters by place-holders with arity; note that no variable appears inside a definition skeleton.

Definitions must obey the following conditions:

1. (variables condition) The variables $x_1, \ldots, x_n, y_1, \ldots, y_m$ are distinct from one another and comprise all of the variables appearing in the definiens; moreover each x_i occurs only once in d.
2. (non overlapping clause) If

   $$d\{x_1, \ldots, x_n\}(y_1) \ldots (y_m) =_{def} e_1$$

   where $x_i - \alpha_i$ var and $y_j - \beta_j$ var, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, and

   $$d\{z_1, \ldots, z_n\}(w_1) \ldots (w_s) =_{def} e_2$$

   where $z_i - \alpha_i$ var and $w_j - \gamma_j$ var, for $i = 1, \ldots, n$ and $j = 1, \ldots, s$, are two definitions with the same definition skeleton then

   - $s = m$
   - $\beta_j = \gamma_j$ for any $j = 1, \ldots, m$

- e_2 is $e_1[z_1, \ldots, z_n, w_1, \ldots, w_s / x_1, \ldots, x_n, y_1, \ldots, y_m]$ where $e[s_1, \ldots, s_k / t_1, \ldots, t_k]$ is the result of the simultaneous textual substitution of the variables t_1, ..., t_k of arities δ_1, ..., δ_k respectively, by the variables s_1, ..., s_k of correspondent arities.

3. (recognizability) The occurrence of a definition skeleton within a string is recognizable.

Remarks

- Condition 2 does not avoid having "duplicate definitions" provided no ambiguity may arise.
- Condition 3 implies the ability to single out within a string all the sub-strings which have been substituted for the place-holders inside the definition skeleton.
- Since in a definition skeleton no variable appears a textual substitution of variables does not modify it.

Examples:

1. Let $x - 0$ var, $f - (0)\ 0$ var, $g - (0)\ 0$ var then

$$[f] \circ [g](x) =_{def} f(g(x))$$

is a definition: $[f] \circ [g]$ is the parametric name, f and g are the name-parameters, $[.] \circ [.]$ is the definition skeleton and x the argument. The intention is to define the function "composition of f and g".

2. Let $t - 0$ var, $x - 0$ var, $y - 0$ var then

$$\mathsf{exchange}[t](x)(y) =_{def} \text{begin } t := x;\ x := y;\ y := t \text{ end}$$

is the definition for a procedure performing the exchange between x and y.

Now we can formally define what an expression of arity α is by giving the rules to form expressions. In the definition we will introduce the predicate $a - \alpha$ exp whose reading is "a is an expression of arity α". A fixed set $\mathcal{D}$ of abbreviating definition is assumed; to stress the dependency on $\mathcal{D}$ the notation $a - \alpha\ \mathsf{exp}_D$ may as well be used.

Definition A.2.2 (Expression of arity α)

1. $$\frac{x - (\alpha_1) \ldots (\alpha_n) 0\ \mathsf{var} \quad a_1 - \alpha_1\ \mathsf{exp} \ldots a_n - \alpha_n\ \mathsf{exp}}{x(a_1) \ldots (a_n) - 0\ \mathsf{exp}} \qquad n \geq 0$$

2. $$\frac{c - (\alpha_1) \ldots (\alpha_n)\ 0\ \mathsf{const} \quad a_1 - \alpha_1\ \mathsf{exp} \ldots a_n - \alpha_n\ \mathsf{exp}}{c(a_1) \ldots (a_n) - 0\ \mathsf{exp}} \qquad n \geq 0$$

3. $$\frac{\begin{array}{l} d\{x_1, \ldots, x_n\}(y_1) \ldots (y_m) =_{def} e \quad e - 0\ \mathsf{exp} \qquad e[\underline{a}, \underline{b}/\underline{x}, \underline{y}] - 0\ \mathsf{exp} \\ x_i - \alpha_i\ \mathsf{var} \quad y_j - \beta_j\ \mathsf{var} \qquad\qquad a_i - \alpha_i\ \mathsf{exp} \quad b_j - \beta_j\ \mathsf{exp} \end{array}}{d[a_1, \ldots, a_n](b_1) \ldots (b_m) - 0\ \mathsf{exp}}$$
$n \geq 0, i = 1, \ldots, n, m \geq 0, j = 1, \ldots, m$

4. $$\frac{b(x) - \beta\ \mathsf{exp} \quad x - \alpha\ \mathsf{var}}{b - (\alpha)\ \beta\ \mathsf{exp}}$$
where the only occurrence of x in $b(x)$ is the manifested one

Notations

- $\underline{a}$ abbreviates $a_1, \ldots, a_n$.
- $e[\underline{a}/\underline{x}]$ is the result of the simultaneous textual substitution of the variables $\underline{x}$ by the expressions $\underline{a}$ within the expression e.
- $d[a_1, \ldots, a_n]$ is a notation to denote the result of the simultaneous textual substitution of the variables $\underline{x}$ by the expressions $\underline{a}$ within the parametric name $d\{x_1, \ldots, x_n\}$. We use this notation to stress that the expressions $\underline{a}$ must be recognizable inside the parametric name.

- $d[a_1, \ldots, a_n]$ and $d[a_1, \ldots, a_n](b_1) \ldots (b_m)$ will be referred to as instance of a parametric name and instance of a definiendum, respectively.

The notions of derivation and theorem are standard. We shall call depth of a derivation the length of the longest branch in the derivation tree.

Remarks

- Rule 4. in the definition A.2.2 requires the ability to recognize where a variable occurs within a string b: we have always supposed to have it. Note that we can inductively define the set of variables that appears inside an expression following the formation rules:

$$\mathsf{V}(x(a_1) \ldots (a_n)) = \{x\} \cup \mathsf{V}(a_1) \cup \ldots \cup \mathsf{V}(a_n)$$
$$\mathsf{V}(c(a_1) \ldots (a_n)) = \mathsf{V}(a_1) \cup \ldots \cup \mathsf{V}(a_n)$$
$$\mathsf{V}(d[a_1, \ldots, a_n](b_1) \ldots (b_m)) = \mathsf{V}(a_1) \cup \ldots \cup \mathsf{V}(a_n) \cup \mathsf{V}(b_1) \cup \ldots \cup \mathsf{V}(b_m)$$
$$\mathsf{V}(b) = \mathsf{V}(b(x))/\{x\}$$

  In our approach we never introduce the notion of free and bound variables but we always deal with variables.

- An expression has one of the following shape

$$x(a_1) \ldots (a_k) \qquad k \geq 0$$
$$c(a_1) \ldots (a_k) \qquad k \geq 0$$
$$d[a_1, \ldots, a_n](b_1) \ldots (b_k) \qquad k \geq 0$$

- The derivation of an expression according to definition A.2.2 is uniquely determinable except for the name of the variable (if rule 4. is applied) and the choice among duplicate definitions. This derives because of the demand to distinguish the occurrences of variables, primitive constants and definition skeletons within a string and, if rule 3. applies, the non overlapping clause assures that an instance of a parametric name always uniquely determines its definition.

- It is easy to see that rules 1. and 2. produce expressions, from old ones, by the use of application under the constrain of correct arity; rule 3. allows the use of abbreviating definition, under the condition that the definiens is an expression; rule 4. is the way to obtain a true abstraction: here this system is different from a typed lambda-calculus where the abstraction is indicated; anyhow in section A.5 we will introduce a set of abbreviating definitions that allow to indicated abstraction.

Expressions are just a syntactic vehicle to denote objects. A definiendum and its definiens must clearly denote the same object and in this sense they have to be considered 'equal'. To achieve this result we extend the textual identity on strings to an equality on expressions of the same arity.

Definition A.2.3 (Definitional equality)

$$1^{=} \quad \frac{x - (\alpha_1) \ldots (\alpha_n)0 \ \mathsf{var} \quad a_1 = b_1 - \alpha_1 \ldots a_n = b_n - \alpha_n}{x(a_1) \ldots (a_n) = x(b_1) \ldots (b_n) - 0} \qquad n \geq 0$$

$$2^{=} \quad \frac{c - (\alpha_1) \ldots (\alpha_n)0 \ \mathsf{const} \quad a_1 = b_1 - \alpha_1 \ldots a_n = b_n - \alpha_n}{c(a_1) \ldots (a_n) = c(b_1) \ldots (b_n) - 0} \qquad n \geq 0$$

$$3_i^{=} \quad \frac{\begin{array}{llll} d\{x_1, \ldots, x_n\}(y_1) \ldots (y_m) =_{def} e & e - 0 \ \mathsf{exp} & e[\underline{a}, \underline{b}/\underline{x}, \underline{y}] = c - 0 & \\ x_i - \alpha_i \ \mathsf{var} \quad y_j - \beta_j \ \mathsf{var} & a_i - \alpha_i \ \mathsf{exp} & b_j - \beta_j \ \mathsf{exp} & \end{array}}{d[a_1, \ldots, a_n](b_1) \ldots (b_m) = c - 0}$$
$$n \geq 0, i = 1, \ldots, n, m \geq 0, j = 1, \ldots, m$$

$$3_{ii}^{=} \quad \frac{\begin{array}{llll} d\{x_1, \ldots, x_n\}(y_1) \ldots (y_m) =_{def} e & e - 0 \ \mathsf{exp} & e[\underline{a}, \underline{b}/\underline{x}, \underline{y}] = c - 0 & \\ x_1 - \alpha_i \ \mathsf{var} \quad y_j - \beta_j \ \mathsf{var} & a_i - \alpha_i \ \mathsf{exp} & b_j - \beta_j \ \mathsf{exp} & \end{array}}{c = d[a_1, \ldots, a_n](b_1) \ldots (b_m) - 0}$$
$$n \geq 0, i = 1, \ldots, n, m \geq 0, j = 1, \ldots, m$$

$$4^{=} \quad \frac{b(x) = d(x) - \beta \quad x - \alpha \ \mathsf{var}}{b = d - (\alpha) \ \beta}$$
where the only occurrence of x in $b(x)$ and $d(x)$ is the manifested one

Remarks

- If we have a derivation of $a = b - \alpha$ then it is possible to prove that $a - \alpha$ exp and $b - \alpha$ exp (the proof is by induction on the depth of the derivation of $a = b - \alpha$)
- It is meaningless to ask for equality between two strings unless they are known to be expressions of the same arity.
- The derivation of definitional equality between two expressions a and b is unique, but for the choice of the name of the variable if rule $4^{=}$ applies and for the choice of (i) or (ii) in rule $3^{=}$ if two instances of a definiendum are considered. This fact can be proved by induction on the 'sum' of the depths of the derivations of a and b.
- It is easy to prove by induction on the depth of the derivation that $=$ is an equivalence relation. Now, by $3_i^{=}$, from $a =_{def} b$ it immediately follows $a = b - 0$.

Theorem A.2.4 (Decidability of the definitional equality) *Let a and b be two expressions of arity α, then the predicate $a = b - \alpha$ is decidable.*

Proof. The proof readily follows by induction on the 'sum' of the depth of the derivations of $a - \alpha$ exp and $b - \alpha$ exp.

Note that the same result could indirectly derive from the existence and unicity of normal form for expressions (see section A.5).

A.3 Some properties of the expressions system

In this section some properties of the system of rules introduced in section A.2 are proved. The main result is theorem A.3.4 which states its closure under substitution.

Theorem A.3.1 *The following property holds for expressions with arity:*

1. *If $x - \alpha$ var then $x - \alpha$ exp.*
2. *If $c - \alpha$ const then $c - \alpha$ exp.*

Proof. The proof of the first point is obtained by induction on the arity of the variable x. After the first point has been proved, the proof of the second point is immediate.

Now we can prove that the system to form expressions is closed under textual substitution. First we will prove a stronger result in the case of substitution of a variable by a variable.

Lemma A.3.2 *If $c - \gamma$ exp, $z - \alpha$ var and $y - \alpha$ var then $c[z/y] - \gamma$ exp is provable with a derivation of the same depth than the derivation of $c - \gamma$ exp.*

Proof. The proof can be obtained by induction on the depth of the derivation of $c - \gamma$ exp. Anyhow, this lemma only states that we can obtain a derivation of $c[z/y] - \gamma$ exp just by substituting all the occurrences of y in the derivation of $c - \gamma$ exp by z: the only condition required is that y and z are variables of the same arity. Attention is required if c is of higher arity $(\gamma_1)\gamma_2$, that is it is obtained, by using rule 4., from $c(x) - \gamma_2$ exp and $x - \gamma_1$ var under the condition that x does not appear in c, and c indeed contains y. In this case it is necessary to avoid x being z: for this reason a rename of x in the derivation of $c(x) - \gamma_2$ exp may be required (allowed by inductive hypothesis).

To obtain the proof of next theorem we need a new definition:

Definition A.3.3 (Arity of a substitution) *Let $b - \beta$ exp, $a_i - \alpha_i$ exp and $x_i - \alpha_i$ var, for $i = 1, \ldots, n$. Then $b[\underline{a}/\underline{x}]$ is obtained from b by a substitution of arity $(\alpha_1) \ldots (\alpha_n)$.*

Now the main theorem on substitution follows.

Theorem A.3.4 (Closure under substitution) *Let $b - \beta$ exp, $a_i - \alpha_i$ exp, $x_i - \alpha_i$ var, for $i = 1, \ldots, n$. Then $b[\underline{a}/\underline{x}] - \beta$ exp.*

Proof. The proof is obtained by double induction: a principal induction on the arity of the performed substitution and a secondary induction on the depth of the derivation of b. Here we deal only with the three most interesting cases.

1. Suppose that

   - $b \equiv y(c_1)\ldots(c_k) - 0$ exp, where $y - (\gamma_1)\ldots(\gamma_k)$ var and $c_j - \gamma_j$ exp, for $j = 1, \ldots, k$,
   - the substitution $[a_1, \ldots, a_n/x_1, \ldots, x_n]$ of arity $(\alpha_1)\ldots(\alpha_n)$ is performed on b
   - y is x_i for some index i

   Then $c_j[\underline{a}/\underline{x}] - \gamma_j$ exp by secondary inductive hypothesis and, for any $i = 1 \ldots n$, $a_i - (\gamma_1)\ldots(\gamma_k)0$ exp by hypothesis. But $a_i - (\gamma_1)\ldots(\gamma_k)0$ exp must have been derived from $a_i(t_1)\ldots(t_k) - 0$ exp, where $t_1, \ldots, t_k$ are distinct variables of arity $\gamma_1, \ldots, \gamma_k$ which do not appear in a_i. Then $a_i(t_1)\ldots(t_k)[\underline{c}[\underline{a}/\underline{x}]/\underline{t}] - 0$ exp follows by principal induction. But $a_i(t_1)\ldots(t_k)[\underline{c}[\underline{a}/\underline{x}]/\underline{t}]$ is $a_i(c_1[\underline{a}/\underline{x}])\ldots(c_k[\underline{a}/\underline{x}])$, since $t_1, \ldots, t_k$ do not appear in a_i, that is $b[\underline{a}/\underline{x}] - 0$ exp.

2. Suppose $b \equiv d[e_1, \ldots, e_k](c_1)\ldots(c_h) - 0$ exp then

   - $d\{v_1, \ldots, v_k\}(t_1)\ldots(t_h) =_{def} g$ is an explicit definition where, for any $i = 1 \ldots k$, $v_i - \eta_i$ var and, for any $j = 1 \ldots h$, $t_j - \gamma_j$ var and all the variables appearing in g are among $v_1, \ldots, v_k, t_1, \ldots, t_h$.
   - $g - 0$ exp
   - $e_1 - \eta_1$ exp, for $i = 1, \ldots, k$
   - $c_j - \gamma_j$ exp, for $j = 1, \ldots, h$
   - $g[\underline{e}, \underline{c}/\underline{v}, \underline{t}] - 0$ exp

   Now the substitution $[a_1, \ldots, a_n/x_1, \ldots, x_n]$ of arity $(a_1)\ldots(a_n)$ is performed on b. Then:

   - $e_i[\underline{a}/\underline{x}] - \eta_i$ exp by secondary inductive hypothesis
   - $c_j[\underline{a}/\underline{x}] - \gamma_j$ exp by secondary inductive hypothesis
   - $g[\underline{e}, \underline{c}/\underline{v}, \underline{t}][\underline{a}/\underline{x}] - 0$ exp by secondary inductive hypothesis

   But $g[\underline{e}, \underline{c}/\underline{v}, \underline{t}][\underline{a}/\underline{x}]$ is $g[\underline{e}[\underline{a}/\underline{x}], \underline{c}[\underline{a}/\underline{x}]/\underline{v}, \underline{t}]$ because of the given hypothesis on the variables that appear in g. Then we can deduce, by using the same definition,

   $$d[e_1[\underline{a}/\underline{x}], \ldots, e_k[\underline{a}/\underline{x}]](c_1[\underline{a}/\underline{x}])\ldots(c_h[\underline{a}/\underline{x}]) - 0 \text{ exp}$$

   that is $b[\underline{a}/\underline{x}] - 0$ exp.

3. Suppose b is an expression of higher arity $(\alpha)\beta$ obtained by application of rule 4 from $b(y) - \beta$ exp and $y - \alpha$ var that does not appear in b, and the substitution $[a_1, \ldots, a_n/x_1, \ldots, x_n]$ of arity $(\alpha_1)\ldots(\alpha_n)$ is performed on b. Now let z be a variable of arity α, different from each x_i, appearing neither in any a_i nor in b. By lemma A.3.2, $b(z) - \beta$ exp can be obtained by using a derivation of the same depth than the one of $b(y) - \beta$ exp; then $b(z)[\underline{a}/\underline{x}] - \beta$ exp by secondary induction hypothesis on $b(z) - \beta$ exp but $b(z)[\underline{a}/\underline{x}]$ is $b[\underline{a}/\underline{x}](z)$ by the choice of z and hence $b[\underline{a}/\underline{x}] - (\alpha)\beta$ exp by rule 4.

By an analogous proof we obtain the following:

Theorem A.3.5 (Substitution of equal expressions) *Let a and b be two expressions of arity α such that $a = b - \alpha$; moreover, let $x_i - \alpha_i$ var and $c_i = d_i - \alpha_i$, for any $i = 1, \ldots, n$. Then $a[\underline{c}/\underline{x}] = b[\underline{d}/\underline{x}] - \alpha$.*

Note that this theorem justifies the term "equality" used in definition A.2.3.

A.4 Decidability of "to be an expression"

A.4.1 New rules to form expressions

In the system so far developed almost every proof proceeds by induction on the depth of the derivation of an expression e, that is when the predicate $e - \epsilon$ exp has a proof. Let us now turn our attention to the decidability of this predicate.

The most natural approach to develop an algorithm to solve this problem is "divide-and-conquer": to decide on a string 'a' first decide on suitable sub-strings of 'a'. Since sub-string property does not hold for rules 3. and 4. in definition A.2.2, the correct development of such an algorithm is not immediate. Point 3 in the remark after definition A.2.2 gives us the right hint: it is convenient to distinguish between "applications", that is the case $k > 0$, and all the other expressions, namely x, c, $d[a_1, \ldots, a_n]$, for $k = 0$. But, this does not suffice because among the premises of rule 3. the "instance" of the definiens is clearly and unavoidably a new string. We can minimize this problem by using a premise which does not depend on the specific instance, as the theorem on closure under substitution suggests to us. This choice will suggest to relate the decidability of expressions to the structure of the set of abbreviating definitions. Hence we begin by showing that the system to form expressions is close under the following rules.

$$(1^*)\quad \frac{x - (\alpha_1)\ldots(\alpha_n)0 \text{ var}}{x - (\alpha_1)\ldots(\alpha_n)0 \text{ exp}}$$

$$(2^*)\quad \frac{c - (\alpha_1)\ldots(\alpha_n)0 \text{ const}}{c - (\alpha_1)\ldots(\alpha_n)0 \text{ exp}}$$

$$(3^*)\quad \frac{\begin{array}{lll} d\{x_1, \ldots, x_n\}(y_1)\ldots(y_m) =_{def} e & & e - 0 \text{ exp} \\ x_i - \alpha_i \text{ var} & y_j - \beta_j \text{ var} & a_i - \alpha_i \text{ exp} \end{array}}{\begin{array}{c} d[a_1, \ldots, a_n] - (\beta_1)\ldots(\beta_n)0 \text{ exp} \\ n \geq 0, i = 1 \ldots n, m \geq 0, j = 1 \ldots m \end{array}}$$

$$(4^*)\quad \frac{b - (\alpha)\beta \text{ exp} \qquad a - \alpha \text{ exp}}{b(a) - \beta \text{ exp}}$$

As regard to rules 1^* and 2^* the result is already stated in theorem A.3.1. For rule 3^* we have to prove that $d[a_1, \ldots, a_n] - (\beta_1)\ldots(\beta_m)$ exp under the assumptions that

1. $d\{x_1, \ldots, x_n\}(y_1)\ldots(y_m) =_{def} e$, where $x_i - \alpha_i$ var and $y_j - \beta_j$ var, is a definition
2. $e - 0$ exp
3. $a_i - \alpha_i$ exp

Let z_1, ..., z_m be variables of arities β_1, ..., β_m respectively which do not appear in any a_i, for $i = 1 \ldots n$. By theorem A.3.1 we know that $z_i - \beta_i$ exp and hence, by theorem A.3.4 on closure under substitution, $e[\underline{a}, \underline{z}/\underline{x}, \underline{y}] - 0$ exp. Thus, by rule 3, $d[a_1, \ldots, a_n](z_1)\ldots(z_m) - 0$ exp and hence, by repeated applications of rule 4, $d[a_1, \ldots, a_n] - (\beta_1)\ldots(\beta_m)$ exp.

For rule 4^* we have to prove that $c_1(c_2) - \beta$ exp if $c_1 - (\alpha)\beta$ exp and $c_2 - \alpha$ exp. But c_1 has higher arity and it must have been derived from $c_1(x) - \beta$ exp, where $x - \alpha$ var is a variable which does not appear in c_1. Now, by closure under substitution and the hypothesis on x, we obtain $c_1(c_2) - \beta$ exp.

Our interest in rules 1^*, 2^*, 3^*, 4^* is in that they suffice to derive any expression. In fact we can prove the following theorem.

Theorem A.4.1 *Let $\mathcal{D}$ be a set of abbreviating definitions. Then, if $b - \beta$ exp then it can be derived by using only rules 1^*, 2^*, 3^*, 4^*.*

Proof. The proof is by induction on the depth of the derivation of $b - \beta$ exp. The result readily follows if b is derived by rule 1. or 2.

If rule 3. is the last rule applied we have that

1. b is $d[a_1, \ldots, a_n](c_1)\ldots(c_m) - 0$ exp

2. $d\{x_1,\ldots,x_n\}(y_1)\ldots(y_m) =_{def} e$, where $x_i - \alpha_i$ var, for $i = 1\ldots n$, and $y_j - \beta_j$ var, for $j = 1\ldots m$, is a definition
3. $e - 0$ exp
4. $e[\underline{a}, \underline{c}/\underline{x}, \underline{y}] - 0$ exp
5. $a_i - \alpha_i$ exp for $i = 1\ldots n$
6. $c_j - \beta_j$ exp for $j = 1\ldots m$

Then, by inductive hypothesis, the following expressions can be derived by using only rules 1^*, 2^*, 3^*, 4^*

1. $e - 0$ exp
2. $a_i - \alpha_i$ exp for $i = 1\ldots n$
3. $c_j - \beta_j$ exp for $i = 1\ldots m$

Then $d[a_1,\ldots,a_n] - (\beta_1)\ldots(b_m)$ exp follows by rule 3^* and hence $b - 0$ exp by rule 4^*.

Finally, if rule 4. is the last rule applied then b is an expression with higher arity, that is $\beta \equiv (\beta_1)\beta_2$. Therefore it has to be $b(x) - \beta_2$ exp with $x - \beta_1$ var which does not appear in b. Now, by inductive hypothesis, $b(x) - \beta_2$ exp can be derived using only rules 1^*, 2^*, 3^*, 4^* and so the last rule applied must have been 4^* and hence it must hold that $b - (\beta_1)\beta_2$ exp is derived using only rules 1^*, 2^*, 3^*, 4^*.

A.4.2 A hierarchy of definitions

The main motivation in pointing out the rules presented in the previous section has been that using only them the decidability of the predicate $a - \alpha$ exp can be readily related to the structure of the set of abbreviating definitions. For this reason from now on we suppose to use only rules 1^*, 2^*, 3^*, 4^* to form expressions. Let us begin by giving the definition of the function lev, that is the level of an expression formed using rules 1^*, 2^*, 3^*, 4^*.

Definition A.4.2 ($\mathsf{lev} : \mathsf{Exps} \to \mathsf{N}$) *The* level *of the expression $e - \alpha$ exp is inductively defined as follows:*

- *if $e \equiv x$, where $x - \alpha$ var, then $\mathsf{lev}(e) = 0$*
- *if $e \equiv c$, where $c - \alpha$ const, then $\mathsf{lev}(c) = 0$*
- *if $e \equiv d[a_1,\ldots,a_n]$ and $\alpha \equiv (\beta_1)\ldots(\beta_m)$, where*
  - *$d\{x_1,\ldots,x_n\}(y_1)\ldots(y_m) =_{def} c$,*
  - *$x_i - \alpha_i$ var, for $i = 1\ldots n$,*
  - *$y_j - \beta_j$ var, for $j = 1\ldots m$,*

  is an abbreviating definition, and $a_i - \alpha_i$ exp, for $i = 1\ldots n$ and $c - 0$ exp then

$$\mathsf{lev}(e) = \mathsf{max}(\mathsf{lev}(a_1),\ldots,\mathsf{lev}(a_n),\mathsf{lev}(c)+1)$$

- *if $e \equiv b(a) - \beta$ exp where*
  - *$b - (\alpha)\beta$ exp and*
  - *$a - \alpha$ exp*

  then $\mathsf{lev}(e) = \mathsf{max}(\mathsf{lev}(b),\mathsf{lev}(a))$

Now we can associate a level to any expression since we have considered all the forms of expressions that can be formed by using rules 1^*, 2^*, 3^*, 4^*. Here, we are particularly interested in those expressions which have the shape of a definiendum, that is $d[x_1, \ldots, x_n](y_1) \ldots (y_m)$. In this case the above definition yields

$$\mathsf{lev}\{d[x_1, \ldots, x_n](y_1) \ldots (y_m)\} = \mathsf{lev}\{c\} + 1$$

if c is the definiens. Therefore the notion of level induces an analogous notion on the subset $\mathcal{D}'$ of abbreviating definitions that can be used to derive expressions (we will call them the *useful definitions*)

$$\mathsf{lev}_{D'}(d[x_1, \ldots, x_n](y_1) \ldots (y_m) =_{def} c) = \mathsf{lev}(c) + 1$$

On the other hand a natural way to try to define an ordering on definitions is to think that a definition must be greater than all the definitions whose skeleton occurs within its definiens. Of course this will not always provide us with an ordering. In the case in which we obtain a well-founded order we call the set $\mathcal{D}$ of abbreviating definitions *well-defined* and we can define the function depth on it:

Definition A.4.3 ($\mathsf{Depth} : \mathsf{Defs} \to \mathsf{N}$) *The* depth *of a definition is inductively defined as follows*

1. $\mathsf{depth}(d[x_1, \ldots, x_n](y_1) \ldots (y_m) =_{def} c) = 1$ *if no definition skeleton appears in* c

2. $\mathsf{depth}(d[x_1, \ldots, x_n](y_1) \ldots (y_m) =_{def} c) = i + 1$ *if* i *is the maximum depth of the definitions whose skeleton appears in* c

Remarks

- $\mathcal{D}'$, i.e. the subset of useful definitions, is well-defined;

- If d is a useful definition, that is if $\mathsf{lev}_D(d)$ is defined, then $\mathsf{lev}_D(d)$ is equal to $\mathsf{depth}(d)$. Indeed, we can prove by induction on the depth of the derivation of an expression e that $\mathsf{lev}(e)$ is equal to the maximum of the depths of the definitions whose skeleton appears in e;

- If the set $\mathcal{D}$ of abbreviating definitions is well-founded and recursively enumerable then there exists a procedure to build up a chain of sets of expressions, such that each one is obtained by adding a new useful definition to the previous one and whose union is the set of all the expressions that can be formed by using $\mathcal{D}$.

The notion of depth can now be extended to strings on an alphabet Σ.

Definition A.4.4 ($\mathsf{depth} : \mathsf{String}(\Sigma) \to \mathsf{N}$) *Let* $\mathcal{D}$ *be a well-defined set of abbreviating definitions and* ω *be a string on an alphabet* Σ *which comprises all of the symbols so far used, then put*

$$\begin{cases} \mathsf{depth}(\omega) = 0 & \text{if no skeleton occurs in } \omega \\ \mathsf{depth}(\omega) = i & \text{if } i \text{ is the maximum among depths of the} \\ & \quad \text{definitions whose skeleton appears in } \omega \end{cases}$$

A.4.3 The algorithm

The decidability of the predicate $\mathsf{exp}_D(e)$ which states that e is an expression, obtained by using only definitions in the a well-defined set of abbreviating definitions $\mathcal{D}$, will be based on the following effective procedure. Let ω be any string on the alphabet Σ.

Algorithm (Decision procedure);

{
input : $\omega \in \mathsf{String}(\Sigma)$,
output : $\omega - \alpha$ exp if ω is an expression of arity α, error otherwise
}
case shape(ω) of
1) a single symbol:
  if $\omega - \alpha$ var (the symbol is a variable)
    then $\omega - \alpha$ exp
  else if $\omega - \alpha$ const (the symbol is a constant)
    then $\omega - \alpha$ exp
  else if $\omega(y_1)\dots(y_m) =_{def} c$,
  where $y_i - \beta_i$ var, for $i = 1, \dots, n$,
  and $c - 0$ exp (the symbol is a parametric name with no parameter)
    then $\omega - (\beta_1)\dots(\beta_m)$ exp
  else error;
2) $b(a)$: (a possible application)
  if $b - (\alpha)\beta$ exp and $a - \alpha$ exp then $\omega - \beta$ exp
  else error;
3) $d[a_1, \dots, a_n]$: (a possible instance of definiendum)
  if $a_i - \alpha_i$ exp for $i = 1, \dots, n$
  and there exists in $\mathcal{D}$ the abbreviating definition
  $d\{x_1, \dots, x_n\}(y_1)\dots(y_m) =_{def} c$,
    where $x_1 - \alpha_i$ var for $i = 1, \dots, n$, $y_j - \beta_j$ var for $j = 1, \dots, m$,
  and $c - 0$ exp
  then $\omega - (\beta_1)\dots(\beta_m)$ exp
  else error;
otherwise error
end;

Let us recall that we assumed that any occurrence of a variable or a constant or a definition skeleton within a string is distinguishable. This is clearly a necessary condition to develop any decision procedure.

The above algorithm decides whether a string ω is an expression or not, that is, we can prove the following theorem.

Theorem A.4.5 (Decidability of $a - \alpha$ exp) *Let $\mathcal{D}$ be a well-defined set of abbreviating definitions. Then, given any string $\omega \in \mathsf{String}(\Sigma)$, the predicate $\omega - \alpha$ exp is decidable.*

Proof. The proof is based on the previous algorithm and consists in showing its total correctness, that is it always terminates and answers $\omega - \alpha$ exp with the correct arity α if ω is an expression, error otherwise. The proof is obtained by principal induction on the depth of ω and subordinate induction on the length of ω.

- Basis (principal induction): $\mathsf{depth}(\omega) = 0$

  1. $\mathsf{length}(w) = 1$. (Basis subordinate induction) ω consists on a single symbol which cannot be a parametric name, thus three possibilities can arise
     - ω is a variable x;
     - ω is a primitive constant c;
     - ω is neither a variable nor a primitive constant.

     In any case the algorithm terminates correctly.
  2. $\mathsf{length}(w) = k + 1$ (Induction step on subordinate induction) If ω is $b(a)$ then the algorithm is recursively applied to both b and a and, by subordinate induction hypothesis, it answers correctly. Clearly we have $\omega - \beta$ exp if and only if $b - (\alpha)\beta$ exp and $a - \alpha$ exp for some arity α. If the shape of ω is not $b(a)$ then error is the correct answer.

- Inductive step principal induction: $\mathsf{depth}(w) = j + 1$

  1. $\mathsf{length}(w) = 1$ (Basis subordinate induction)
     The only positive possibility is that ω is a parametric name d. Then $\omega - (\beta_1) \dots (\beta_m)$ exp, for $m \geq 0$ if and only if $\omega(y_1) \dots (y_m) =_{def} c$, where $y_i - \beta_i$ var for $i = 1, \dots, m$, and $c - 0$ exp. Since $\mathsf{depth}(c) \leq j$ the algorithm correctly terminates. Otherwise error is the correct answer.
  2. $\mathsf{length}(w) = k + 1$ (Induction step subordinate induction)
     There are only three possibilities:
     - ω is $b(a)$. Since the inductive hypothesis holds both for b and a the case is analogous to the above step (2).
     - ω is $d[a_1, \dots, a_n]$. In this case

       $$\omega - (\beta_1) \dots (\beta_m) \text{ exp}$$

       for some $m \geq 0$ if and only if

       $$a_i - \alpha_i \text{ exp}$$

       for $i = 1 \dots n$ and

       $$d\{x_1, \dots, x_n\}(y_1) \dots (y_m) =_{def} c$$

       where $x_i - \alpha_i$ var for $i = 1, \dots, n$ and $y_h - \beta_h$ var for $h = 1, \dots, m$, is an abbreviating definition in $\mathcal{D}$ and $c - 0$ exp.
       Now, $\mathsf{depth}(c) \leq j$, $\mathsf{depth}(a_i) \leq j + 1$ and $\mathsf{length}(a_i) < k + 1$ for all $i = 1, \dots, n$. Then the recursive calls on c and a_i, by inductive hypothesis, must terminate in a correct way and hence the answer will be correct
     - the shape of ω is neither $b(a)$ nor $d[a_1, \dots, a_n]$ and therefore error is the correct answer.

A.5 Abstraction and normal form

It is easy to convince ourselves that by using the rules of definition A.2.2, in order to obtain non saturated expressions we can either start from variables or constants of higher arity or use some abbreviating definition. In standard lambda-calculus, non saturated expressions are built up by λ-abstraction. We can show how a suitable set of abbreviating definitions can play the role of λ-abstraction. To this aim we will introduce all the abbreviating definitions represented by the following abbreviating definition schema:

Definition A.5.1 (Abstraction schema)

$$((\mathbf{x})\ c^{\mathbf{x}})(x)(y_1) \dots (y_m) =_{def} c(y_1) \dots (y_m)$$

where c stands for any expression of arity $(\beta_1) \dots (\beta_m)$ such that no variable, except x, occurs in c more than once, $\mathbf{x}$ is a new symbol (not a variable!) used to recall the variable x (we will call $\mathbf{x}$ the ghost *of x) and $c^{\mathbf{x}}$ is the string obtained from c by substituting each occurrence of x by $\mathbf{x}$ and each occurrence of a variable w by $[w]$.*

Any instance of this schema, obtained by choosing a variable x of arity α, an expression c of arity $(\beta_1) \dots (\beta_m)$ such that no variable, except x, occurs in c more than once and the two by two distinct variables $y_1, \dots, y_m$, for $m \geq 0$ of arities $\beta_1, \dots, \beta_m$ which do not appear in c, turns out to be a definition. The parametric name is $((\mathbf{x})\ c^{\mathbf{x}})$, the name-parameters are all the variables appearing in the expression c except x, which will be called the *abstracted variable*. In fact, condition 1. on abbreviating definitions is satisfied because of the choice of $y_1, \dots, y_m$, the fact that all the variables occurring in c occur also in $c^{\mathbf{x}}$, except x that appears explicitly in the definiendum and the requirements on the occurrences of the variables in c. The validity of condition 2. follows from the fact that the function $\{\}^{\mathbf{x}}$ is injective on the set of the expressions, i.e., whenever c and e are expressions, if $c^{\mathbf{x}}$ and $e^{\mathbf{x}}$ are equal strings then also c and e are equal

strings. Concerning condition 3., let us note that the skeleton of an abstraction always starts with "((**y**)" and ends with ")", hence to find out the first outermost occurrence of an abstraction skeleton within a string we can use the following procedure. Scan the string until "((**y**)" is found for some **y**. It is the beginning of a possible skeleton. From now on continue the scanning and count both the number of non completed abstractions (i.e. "((**x**)" for some **x** met but the matching ")" not yet met) and the number of pending open square brackets. When these two quantities become equal skip the subsequent sub-string till the next matching closed square bracket. The skipped sub-string is not part of the abstraction skeleton but it is one of the possible parameters. The skeleton ends when the ")" matching with "((**y**)" is found.

Example:
Let $((\mathbf{y})((\mathbf{x})f(\mathbf{x})([\mathbf{y}]))([+([x])(\mathbf{y})]))$ be the string to be tested, then

```
((y)   ((x)  f(x) ([y])) ([  + ([x]) (y)]))
|      |           |      |     |
ast=1  ast=2       ast=2  ast=2 ast=2
#[ =0  #[ =0       #[ =1  #[ =1 #[ =2
```

and hence the skeleton is $((\mathbf{y})((\mathbf{x})f(\mathbf{x})([\mathbf{y}]))([+([\ ])(\mathbf{y})]))$

Let us give you another example of an instance of the abstraction schema.

Example:
Let $x - 0$ var, $y - 0$ var, $+ - (0)(0)0$ const, then the expression $+(x)(y) - 0$ can be formed and hence

1. $((\mathbf{x}) + (\mathbf{x})([y]))(x) =_{def} +(x)(y)$
   is an instance of the abstraction schemata. The parametric name is
   $$((\mathbf{x}) + (\mathbf{x})([y]))$$
   and y is the only name parameter. This definition can be used to derive by rule 3
2. $((\mathbf{x}) + (\mathbf{x})([a(x)])) - 0$ exp
   provided $a(x) - 0$ exp.

Note that in (2) we can single out the sub-string $a(x)$, i.e. the parameter, and that the variable x which occurs in the expression $a(x)$ occurs also in the expression $((\mathbf{x}) + (\mathbf{x})([a(x)]))$.

It is easy to see that the main difference between our approach and the one of typed lambda-calculus, about the way to treat abstraction, is in that there one keeps track of the abstracted variables while here one keeps track of the variables that remain (free) inside an expression: in this way we have no need to introduce the notion of free and bound variables and to operate any α-conversion when abstracting or substituting.

Note that the requirement on the occurrences of variables inside the expression c is not severe since, at the expression's level, we loose nothing. In fact, suppose c is any expression, y a variable and $x_1, \ldots, x_n$ is the list of (possibly) non-distinct variables formed by taking all the variables occurring in c, except y, in the left-to-right order (possibly) with multi-occurrences. It easy to convince ourselves that there always exists another expression c', where no variable, except y, occurs more than once, such that $c'[x_1, \ldots, x_n/w_1, \ldots, w_n]$ is c if $w_1, \ldots, w_n$ is the ordered list of variables occurring in c'. Then even if in c some variable occurs more than once (so that the definition $((\mathbf{y})\ c^{\mathbf{y}})(y)(z_1)\ldots(z_k) =_{def} c(z_1)\ldots(z_k)$ is not allowed) there is always another expression c' such that

$$((\mathbf{y})\ c'^{\mathbf{y}})(y)(z_1)\ldots(z_k) =_{def} c'(z_1)\ldots(z_k)$$

is allowed. Then the expression $((\mathbf{y})\ c^{\mathbf{y}})$, which is $((\mathbf{y})\ c'^{\mathbf{y}})[x_1, \ldots, x_n]$, can be formed.

Example:

$$((\mathbf{x})\ \mathbf{x}([y])([y]))$$

is an expression obtained by using the definition

$$((\mathbf{x})\ \mathbf{x}([y_1])([y_2]))(x) =_{def} x(y_1)(y_2)$$

by substituting the name parameters y_1, y_2 with the arguments y and y.

Moreover we can prove that, supposing $c - (\gamma_1) \dots (\gamma_k)$ exp,

$$((\mathbf{y})c^{\mathbf{y}})(y) = c - (\gamma_1) \dots (\gamma_k)$$

holds. In fact, let $z_j - \gamma_j$ var, for $j = 1 \dots k$, be new variables; then, since the expression $c'(z_1) \dots (z_k)[x_1, \dots, x_n/w_1, \dots, w_n]$ is $c(z_1) \dots (z_k)$, by applying rule $3^=$ and theorem A.3.5 on substitution of equal expressions, we obtain

$$((\mathbf{y})\ c'^{\mathbf{y}})(y)(z_1) \dots (z_k)[x_1, \dots, x_n/w_1, \dots, w_n] = c(z_1) \dots (z_k) - 0$$

but the left side member of this equality is exactly

$$((\mathbf{y})c^{\mathbf{y}})(y)(z_1) \dots (z_k)$$

Hence, by iterated applications of rule $4^=$, we obtain

$$((\mathbf{y})\ c^{\mathbf{y}})(y) = c - (\gamma_1) \dots (\gamma_k)$$

Summarizing, we have the following:

Lemma A.5.2 *Let* $c - (\gamma_1) \dots (\gamma_k)$ exp *and* $y - \alpha$ var*, then*

1. $((\mathbf{y})\ c^{\mathbf{y}}) - (\alpha)(\gamma_1) \dots (\gamma_k)$ exp *in which* y *does not appear*
2. $((\mathbf{y})\ c^{\mathbf{y}})(y) = c - (\gamma_1) \dots (g_k)$

Another consequence of the above consideration is that, if

$$((\mathbf{y})\ c^{\mathbf{y}})[a_1, \dots, a_n] - (\beta_1) \dots (\beta_k)$$

is an expression which is an instance of an abstraction definition, then

$$((\mathbf{y})\ c[a_1[t/y], \dots, a_n[t/y]/w_1, \dots, w_n]^{\mathbf{y}}) - (\beta_1) \dots (\beta_k)$$

where t is a new variable and w_1, ..., w_n are all the variables occurring in c, except y, is an expression. Moreover, since an instance of a definition is always definitionally equal to its definiens correctly substituted we have also that

$$((\mathbf{y})\ c^{\mathbf{y}})[a_1, .., a_n] = ((\mathbf{y})\ c[a_1[t/y], .., a_n[t/y]/w_1, .., w_n]^{\mathbf{y}})[y/t] - (\beta_1)..(\beta_k)$$

Example:

$$((\mathbf{x})\ \mathbf{x}([x(z)])) - (\alpha)0 \text{ exp}$$

is an expression obtained by means of the definition

$$((\mathbf{x})\ \mathbf{x}([y]))(x) =_{def} x(y)$$

Hence

$$((\mathbf{x})\ x(y)[x(z)[t/x]/y]^{\mathbf{x}}) - (\alpha)0,$$

which coincides with

$$((\mathbf{x})\ \mathbf{x}([t]([z]))) - (\alpha)0,$$

is an expression, that can be formed by means of the definition

$$((\mathbf{x})\ \mathbf{x}([w_1]([w_2])))(x) =_{def} x(w_1(w_2))$$

Then

$$((\mathbf{x})\ \mathbf{x}([t(z)])) = ((\mathbf{x})\ \mathbf{x}([t]([z]))) - (\alpha)0$$

since

$$x(y)[x(z), s/y, x] \equiv x(w_1(w_2))[x, z, s/w_1, w_2, x]$$

where $s - (\alpha)0$ var is a new variable.

From now on, we assume to derive expressions within a system on a set $\mathcal{D}$ of definitions which comprises all of the instances of the abstraction schema, unless otherwise stated.

Remark

The use of the above schema of abbreviating definition does not affect the decidability of the predicate $a - \alpha$ exp. In fact, the only problem can derive from the fact that at point 3 of the decision algorithm we require the ability to recognize an instance of a definiendum. So, let us suppose that $((\mathbf{x})\ b)$ is the string which must be analyzed. We already showed how to single out the definition skeleton. The possible parameter $a_1, \ldots, a_n$ are just the sub-strings skipped in that process. They must be first substituted inside the skeleton by new variables $x_1, \ldots, x_n$ whose arity is determined by the arity of the expressions $a_1, \ldots, a_n$. Then, to single out the definition (indeed we will find out one definition among a set of duplicate definitions) we have first to invert the function $\{\}^{\mathbf{x}}$. This can be done by cutting off the square brackets immediately around the new variables and by substituting $\mathbf{x}$, that is the ghost of x, by x unless it occurs within the skeleton of an inner abstraction with the same ghost. Secondly we have to decide on the resulting string which is "simpler" than the initial one. If it is an expression a of arity $(\beta_1)\ldots(\beta_m)$ the definition we were looking for is $((\mathbf{x})\ b)(x)(y_1)\ldots(y_m) =_{def} a(y_1)\ldots(y_m)$.

Example:

Suppose that $x - 0$ var, $f - (0)((0))$ const, $+ - (0)(0)$ const and that we want to analyze the string

$$((\mathbf{x})\ ((\mathbf{x})\ f(\mathbf{x})([[(\mathbf{x})\ +(\mathbf{x})([\mathbf{x}])])])).$$

First we look for the skeleton of the outermost abstraction "$((\mathbf{x})\ \ldots)$": it is the entire string

$$((\mathbf{x})\ ((\mathbf{x})\ f(\mathbf{x})([[(\mathbf{x})\ +(\mathbf{x})([\mathbf{x}])])]))$$

Now, we must invert the function $\{\ \}^{\mathbf{x}}$ on

$$((\mathbf{x})\ f(\mathbf{x})([[(\mathbf{x})\ +(\mathbf{x})([\mathbf{x}])])])$$

and we obtain the string

$$((\mathbf{x})\ f(\mathbf{x})([[(\mathbf{x})\ +(\mathbf{x})([x])])])$$

Then we recursively continue by analyzing this resulting string and finally we obtain

$$((\mathbf{x})\ f(\mathbf{x})([[(\mathbf{x})\ +(\mathbf{x})([x])])]) - (0)\ \mathsf{exp}$$

Therefore the definition we are looking for is

$$((\mathbf{x})\ ((\mathbf{x})\ f(\mathbf{x})([[(\mathbf{x})\ +(\mathbf{x})([\mathbf{x}])])]))(x)(s) =_{def} ((\mathbf{x})\ f(\mathbf{x})([[(\mathbf{x})\ +(\mathbf{x})([x])])])(s)$$

A.5.1 α, β, η and ξ conversion

When all the instances of the abstraction schema had been introduced, we can prove the validity of equality analogous to those introduced by α, β, η, ξ reductions of lambda-calculus. In fact, after lemma A.5.2, we know that, for any expression c of arity γ and any variable x of arity α, $((\mathbf{x})\ c^{\mathbf{x}})(x) = c - \gamma$. Hence

Theorem A.5.3 (α-equality) *Suppose $c - \gamma$ exp and $y - \alpha$ var is a variable which does not appear in c. Then*

$$\textit{(α-equality)} \qquad ((\mathbf{x})\ c^{\mathbf{x}}) = ((\mathbf{y})\ c[y/x]^{\mathbf{y}}) - (\alpha)\gamma$$

Proof. By lemma A.5.2

$$((\mathbf{y})\ c[y/x]^{\mathbf{y}})(y) = c[y/x] - \gamma$$

Then by the theorem on substitution of equal expressions

$$((\mathbf{y})\ c[y/x]^{\mathbf{y}})(y)[x/y] = c[y/x][x/y] - \gamma$$

which yields

$$((\mathbf{y})\ c[y/x]^{\mathbf{y}})(x) = c - \gamma$$

since y appears neither in $((\mathbf{y})\ c[y/x]^{\mathbf{y}})$ nor in c.

But $c = ((\mathbf{x})\ c^{\mathbf{x}})(x) - \gamma$ and hence $((\mathbf{y})\ c[y/x]^{\mathbf{y}})(x) = ((\mathbf{x})\ c^{\mathbf{x}})(x) - \gamma$ by transitivity. Thus we finally obtain

$$((\mathbf{y})\ c[y/x]^{\mathbf{y}}) = ((\mathbf{x})\ c^{\mathbf{x}}) - (\alpha)\gamma$$

by rule $4^{=}$, since x appears neither in $((\mathbf{x})\ c^{\mathbf{x}})$ nor in $((\mathbf{y})\ c[y/x]^{\mathbf{y}})$.

Theorem A.5.4 (β-equality) *Suppose* $c - \gamma$ exp *and* $a - \alpha$ exp. *Then*

$$\textit{(β-equality)} \qquad ((\mathbf{x})\ c^{\mathbf{x}})(a) = c[a/x] - \gamma$$

Proof. The result follows directly from lemma A.5.2 and theorem A.3.5.

Theorem A.5.5 (η-equality) *Let* $c - (\alpha)\gamma$ exp *and* $x - \alpha$ var *which does not appear in* c. *Then*

$$\textit{(η-equality)} \qquad ((\mathbf{x})\ c(x)^{\mathbf{x}}) = c - (\alpha)\gamma$$

Proof. By lemma A.5.2 we obtain that

$$((\mathbf{x})\ c(x)^{\mathbf{x}})(x) = c(x) - \gamma$$

Hence by rule $4^{=}$

$$((\mathbf{x})\ c(x)^{\mathbf{x}}) = c - (\alpha)\gamma$$

since x appears neither in c nor in $((\mathbf{x})\ c(x)^{\mathbf{x}})$.

Theorem A.5.6 (ξ-equality) *Suppose that* $b - \gamma$ exp, $d - \gamma$ exp, $x - \alpha$ var *and that* $b = d - \gamma$. *Then*

$$\textit{(ξ-equality)} \qquad ((\mathbf{x})\ b^{\mathbf{x}}) = ((\mathbf{x})\ d^{\mathbf{x}}) - (\alpha)\gamma$$

Proof. By lemma A.5.2

$$((\mathbf{x})\ b^{\mathbf{x}})(x) = b - \gamma$$

and

$$((\mathbf{x})\ d^{\mathbf{x}})(x) = d - \gamma$$

Thus, by transitivity,

$$((\mathbf{x})\ b^{\mathbf{x}})(x) = ((\mathbf{x})\ d^{\mathbf{x}})(x) - \gamma$$

and finally

$$((\mathbf{x})\ b^{\mathbf{x}}) = ((\mathbf{x})\ d^{\mathbf{x}}) - (\alpha)\gamma$$

by rule $4^{=}$ since x appears neither in $((\mathbf{x})\ b^{\mathbf{x}})$ nor in $((\mathbf{x})\ d^{\mathbf{x}})$.

A.5.2 Normal form

Let us introduce the concept of normal form for an expression. The definition will be given by induction on the construction of the expression.

Definition A.5.7 (Normal form) *Let a be an expression. Then*

1. *If a has arity 0 then it is in normal form if its shape is*

$$f(b_1)\dots(b_m)$$

where f is a variable or a primitive constant of arity $(\beta_1)\dots(\beta_m)0$ and, for any $j = 1\dots m$, b_j is an expression in normal form of arity β_j.

2. *If a has arity $(\alpha)\beta$ then it is in normal form if its shape is*

$$((\mathbf{x})\ c^{\mathbf{x}})$$

where $x - \alpha$ var and c is an expression of arity β in normal form.

Note that, given any expression e, there always exists at least an expression $\mathsf{nf}(e)$ in normal form which is definitional equivalent to e. The following algorithm can be used to find it out.

Algorithm (Normalization algorithm)

1. Suppose that the arity of e is $(\alpha)\beta$. Then it has been obtained by an application of rule 4. Now, let x be a variable of arity α which does not appear in e. Then

$$\mathsf{nf}(e) \equiv ((\mathbf{x})\ \mathsf{nf}(e(x))^{\mathbf{x}})$$

2. Suppose that the arity of e is 0 and that it has been obtained by using rule 3, that is

$$e \equiv d[a_1,\dots,a_n](b_1)\dots(b_m)$$

for some abbreviating definition $d\{x_1,\dots,x_n\}(y_1)\dots(y_m) =_{def} c$. Then

$$\mathsf{nf}(e) \equiv \mathsf{nf}(c[\underline{a},\underline{b}/\underline{x},\underline{y}])$$

3. Suppose that the arity of e is 0 and that it has been obtained by using the rule 1 or 2, that is, $e \equiv f(b_1)\dots(b_m)$, where f is a variable or a primitive constant of arity $(\beta_1)\dots(\beta_m)$ and $b_j - \beta_j$ exp, for $j = 1\dots m$. Then

$$\mathsf{nf}(e) \equiv f(\mathsf{nf}(b_1))\dots(\mathsf{nf}(b_m))$$

The proof that this algorithm preserves definitional equality is obtained by induction on the depth of the derivation of the expression e. A sketch of the proof is the following:

Case 1. By theorem A.5.2

$$((\mathbf{x})\ \mathsf{nf}(e(x))^{\mathbf{x}})(x) = \mathsf{nf}(e(x)) - \beta$$

But, by inductive hypothesis, $\mathsf{nf}(e(x)) = e(x) - \beta$ and so, by transitivity and rule 4,

$$((\mathbf{x})\ \mathsf{nf}(e(x))^{\mathbf{x}}) = e - (\alpha)\beta$$

Case 2. By inductive hypothesis

$$\mathsf{nf}(c[\underline{a},\underline{b}/\underline{x},\underline{y}]) = c[\underline{a},\underline{b}/\underline{x},\underline{y}] - 0$$

but, by rule $3_i^=$,

$$c[\underline{a},\underline{b}/\underline{x},\underline{y}] = d[a_1,\dots,a_n](b_1)\dots(b_m) - 0$$

Case 3. Immediate by inductive hypothesis.

Moreover the algorithm always terminates because at each step a 'simpler' (with respect to derivation) expression is considered. Finally it is straightforward to check that the final shape is the required one.

It is easy to see that an expression in normal form contain no definition, a part abstraction, hence we can state the following theorem about definition elimination.

Theorem A.5.8 (Definition elimination) *Let a be any expression. Then, there exists an expression $\mathsf{nf}(a)$, equivalent to a, in which no definition appears, except abstraction.*

We can also prove that the normal form is a good representative for the class of definitional equivalent expressions. Let us first prove the following lemma.

Lemma A.5.9 *If Π is the proof of $((\mathbf{x})\ c^{\mathbf{x}}) = ((\mathbf{y})\ e^{\mathbf{y}}) - (\alpha)\beta$ then there is a proof Π^* of $c[t/x] = e[t/y] - \beta$, whose depth is less that the depth of Π.*

The proof consist just in constructing Π^*, by analyzing the proof Π.

Theorem A.5.10 (Unicity of normal form) *Let a and b be two expressions in normal form. Then $a = b - \alpha$ implies that a and b have the same pattern, that is, they differ only for the name of the abstracted variables.*

Proof. The proof proceeds by induction on the depth of the derivation of $a = b - \alpha$, using the previous lemma in the case two abstractions are considered. Now the following corollary easily follows.

Corollary A.5.11 *Two expressions are definitionally equivalent if and only if they have the same normal form (except for the name of abstracted variables).*

A.6 Relation with typed λ-calculus

In this section we examine the relationship between our expressions and standard typed lambda-calculus. We will consider expressions on a set of abbreviating definitions which comprise only all the instances of the abstraction schema. We will not give complete proofs of the stated theorems since they are standard or can easily be reconstructed.

Since a lot of slightly different formulations of typed lambda-calculus can be found in the literature, here we expose the one that we will use and some of its properties we are interested in.

Definition A.6.1 (System^λ) *We will call System^λ the set of strings which we obtain by the following inductive definition:*

$$1_\lambda \quad \frac{x - (\alpha_1)\dots(\alpha_n)\ \mathsf{var}}{x - (\alpha_1)\dots(\alpha_n)\ \mathsf{exp}^\lambda} \qquad \mathsf{FV}(x) = \{x\}$$

$$2_\lambda \quad \frac{c - (\alpha_1)\dots(\alpha_n)\ \mathsf{const}}{c - (\alpha_1)\dots(\alpha_n)\ \mathsf{exp}^\lambda} \qquad \mathsf{FV}(c) = \emptyset$$

$$3_\lambda \quad \frac{b - (\alpha)\beta\ \mathsf{exp}^\lambda \qquad a - \alpha\ \mathsf{exp}^\lambda}{b(a) - \alpha\ \mathsf{exp}^\lambda} \qquad \mathsf{FV}(b(a)) = \mathsf{FV}(b) \cup \mathsf{FV}(a)$$

$$4_\lambda \quad \frac{b - \beta\ \mathsf{exp}^\lambda \qquad x - \alpha\ \mathsf{var}}{((x)\ b) - (\alpha)\beta\ \mathsf{exp}^\lambda} \qquad \mathsf{FV}(((x)b)) = \mathsf{FV}(b) - \{x\}$$

We will use the convention to use different names for different variables: by this convention we will avoid to say "let $x - \alpha$ var and $y - \alpha$ var and $x \neq y \dots$".

The substitution of a variable with a λ-expression within a λ-expression is defined in the usual way.

Definition A.6.2 (Substitution :=) *Let* $x - \alpha$ var *and* $e - \alpha$ exp^λ. *Then*

$$
\begin{array}{rcl}
x[x := e] & \equiv & e \\
y[x := e] & \equiv & y, \textit{where } y - \beta \ \mathsf{var} \\
c[x := e] & \equiv & c, \textit{where } c - \beta \ \mathsf{const} \\
b(a)[x := e] & \equiv & b[x := e](a[x := e]) \\
((x)\ b)[x := e] & \equiv & ((x)\ b) \\
((y)\ b)[x := e] & \equiv & ((t)\ b[y := t][x := e]), \\
 & & \textit{where } t \textit{ is a new variable of the same arity of } y
\end{array}
$$

Remarks

- Let Π be the derivation of $b - \beta$ exp^λ. Then the proof Σ of $b[x := t] - \beta$ exp^λ, where x and t are two variables of the same arity, has the same complexity than Π. Note that this remark justify the above definition A.6.2.
- Let Π be the derivation of $((x)\ b) - (\alpha)\beta$ exp^λ. Then the proof Σ of $((t)\ b[x := t]) - (\alpha)\beta$ exp^λ, where x and t are two variables of the same arity, has the same complexity than Π.

We recall the predicate of equality between two *lambda*-expressions induced by $\beta\eta$-conversion.

Definition A.6.3 (λ-equality) *The equality between two λ-expressions is the minimal relation between two λ-expressions which satisfies the following conditions:*

$$
\begin{array}{ll}
1^=_\lambda & \dfrac{x - (\alpha_1) \ldots (\alpha_n)\ \mathsf{var}}{x =_\lambda x - (\alpha_1) \ldots (\alpha_n)} \\[2ex]
2^=_\lambda & \dfrac{c - (\alpha_1) \ldots (\alpha_n)\ \mathsf{const}}{c =_\lambda c - (\alpha_1) \ldots (\alpha_n)} \\[2ex]
3^=_\lambda & \dfrac{b_1 =_\lambda b_2 - (\alpha)\beta \quad a_1 =_\lambda a_2 - \alpha}{b_1(a_1) =_\lambda b_2(a_2) - \alpha} \\[2ex]
\alpha^=_\lambda & \dfrac{b - \beta\ \mathsf{exp}^\lambda \quad x - \alpha\ \mathsf{var} \quad y - \alpha\ \mathsf{var}}{((x)b) =_\lambda ((y)b[x := y]) - (\alpha)\beta} \\
 & \textit{provided } y \textit{ does not appear in } b \\[2ex]
\beta^=_\lambda & \dfrac{b - \beta\ \mathsf{exp}^\lambda \quad x - \alpha\ \mathsf{var} \quad a - \alpha\ \mathsf{exp}^\lambda}{((x)b)(a) =_\lambda b[x := a] - \beta} \\[2ex]
\xi^=_\lambda & \dfrac{b =_\lambda d - \beta\ \mathsf{exp}^\lambda \quad x - \alpha\ \mathsf{var}}{((x)b) =_\lambda ((x)d) - (\alpha)\beta} \\[2ex]
\eta^=_\lambda & \dfrac{b - (\alpha)\beta\ \mathsf{exp}^\lambda \quad x - \alpha\ \mathsf{var}}{((x)b(x)) =_\lambda b - (\alpha)\beta} \\
 & \textit{provided } x \textit{ does not appear in } b \\[2ex]
(ref_\lambda) & \dfrac{b - \beta\ \mathsf{exp}^\lambda}{b =_\lambda b - \beta} \\[2ex]
(simm_\lambda) & \dfrac{a =_\lambda b - \beta}{b =_\lambda a - \beta} \\[2ex]
(trans_\lambda) & \dfrac{a =_\lambda b - \beta \quad b =_\lambda c - \beta}{a =_\lambda c - \beta}
\end{array}
$$

Some well-known properties of the substitution := are the following.

Lemma A.6.4 *The following properties hold*

1. *If* $b - \beta\ \mathsf{exp}^\lambda$, $x - \alpha\ \mathsf{var}$,
   then $b[x := x] =_\lambda b - \beta$;

2. *If* $a =_\lambda b - \beta\ \mathsf{exp}^\lambda$, $x - \alpha\ \mathsf{var}$ *and* $c - \alpha\ \mathsf{exp}^\lambda$,
   then $a[x := c] =_\lambda b[x := c] - \beta$;

3. *If* $x - \alpha\ \mathsf{var}$ *does not appear in* $b - \beta\ \mathsf{exp}^\lambda$ *and* $c - \alpha\ \mathsf{exp}^\lambda$,
   then $b[x := c] =_\lambda b - \beta$;

4. *(Substitution lemma) If* $y - \gamma\ \mathsf{var}$ *does not appear in* $a - \alpha\ \mathsf{exp}^\lambda$ *and* $b - \beta\ \mathsf{exp}^\lambda$, $c - \gamma\ \mathsf{exp}^\lambda$ *and* $x - \alpha\ \mathsf{var}$,
   then $b[y := c][x := a] =_\lambda b[x := a][y := c[x := a]] - \beta$;

5. *If* $y - \beta\ \mathsf{var}$ *does not appear in* $a - \alpha\ \mathsf{exp}^\lambda$ *and* $x - \alpha\ \mathsf{var}$,
   then $((y)\ d)[x := a] \equiv ((y)\ d[x := a])$
   (remember the convention on the variable names!);

6. *If, for any* $i = 1 \ldots n$, $w_i - \gamma_i\ \mathsf{var}$ *does not appear in* $a - \alpha\ \mathsf{exp}^\lambda$, $b - \beta\ \mathsf{exp}^\lambda$, $c_i - \gamma_i\ \mathsf{exp}^\lambda$, *for any* $i = 1 \ldots n$, *and* $x - \alpha\ \mathsf{var}$
   then
   $$b[w_1 := c_1] \ldots [w_n := c_n][x := a] =_\lambda b[x := a][w_1 := c_1[x := a]] \ldots [w_n := c_n[x := a]] - \beta$$

Our aim is now to map every expression into a λ-expression and vice-verse by using maps that preserve equality. Let us begin with the following definition.

Definition A.6.5 ($\mathsf{F} : \mathsf{System}^\lambda \to \mathsf{Exp}$) *The map* F *between* System^λ *and* Exp *is inductively defined as follows:*

$$\mathsf{F}(x) \equiv x, \textit{where } x - \alpha\ \mathsf{var}$$

$$\mathsf{F}(c) \equiv c, \textit{where } c - \alpha\ \mathsf{const}$$

$$\mathsf{F}(b(a)) \equiv \mathsf{F}(b)(\mathsf{F}(a))$$

$$\mathsf{F}(((x)\ b)) \equiv ((\mathbf{x})\ \mathsf{F}(b)^{\mathbf{x}})$$

Theorem A.6.6 *If* $b - \beta\ \mathsf{exp}^\lambda$ *then* $\mathsf{F}(b) - \beta\ \mathsf{exp}$.

Proof. The proof is by induction on the construction of b in the System^λ.

Moreover, it is easy to see that supposing $b - \beta\ \mathsf{exp}^\lambda$, $\mathsf{FV}(b) = \mathsf{V}(\mathsf{F}(b))$ can be proved.

The next theorem states the fundamental result that the map F respects equality. Its proof needs, beside the result on α, β, η and ξ equalities that we showed in the previous section, also the following lemma.

Lemma A.6.7 *If* $a - \beta\ \mathsf{exp}^\lambda$, $x - \alpha\ \mathsf{var}$ *and* $c - \alpha\ \mathsf{exp}^\lambda$ *then the map* F *preserves substitution, that is,* $\mathsf{F}(a[x := c]) = \mathsf{F}(a)[\mathsf{F}(c)/x] - \beta$.

Proof. The proof is by induction on the derivation of $a - \beta\ \mathsf{exp}^\lambda$.

Theorem A.6.8 *If* $a =_\lambda b - \beta$ *then* $\mathsf{F}(a) = \mathsf{F}(b) - \beta$

Proof. The proof is by induction on the derivation of $a =_\lambda b - \beta$.

The map in the opposite direction, that is from Exp into System^λ, is defined on expressions constructed by the rules 1^*, 2^*, 3^* and 4^* the we introduced in section A.3.

Definition A.6.9 ($\mathsf{G} : \mathsf{Exp} \to \mathsf{System}^\lambda$) *The map* G *between* Exp *and* System^λ *is inductively defined as follows:*

$$\mathsf{G}(x) \equiv x, \textit{where } x - \alpha \ \mathsf{var}$$

$$\mathsf{G}(c) \equiv c, \textit{where } c - \alpha \ \mathsf{const}$$

$$\mathsf{G}(b(a)) \equiv \mathsf{G}(b)(\mathsf{G}(a))$$

$$\mathsf{G}(((\mathbf{x})\ b^{\mathbf{x}})[a_1, \ldots, a_n]) \equiv ((x)\ \mathsf{G}(b))[w_1 := \mathsf{G}(a_1)] \ldots [w_n := \mathsf{G}(a_n)]$$

where a_1, ..., a_n *are the actual parameters which substitute the name parameters* w_1, ..., w_n.

Theorem A.6.10 *If* $b - \beta \ \mathsf{exp}$ *then* $\mathsf{G}(b) - \beta \ \mathsf{exp}^\lambda$

Proof. The proof is by induction on the construction of the expression b by using rules 1^*, 2^*, 3^* and 4^*.

The next lemma is analogous to the previous lemma A.6.7.

Lemma A.6.11 *Let* $e - \beta \ \mathsf{exp}$, $x_1 - \alpha_i \ \mathsf{var}$ *and* $a_i - \alpha_i \ \mathsf{exp}$, *for* $i = 1 \ldots n$, *such that no* x_i *appears in any* a_j. *Then*

$$\mathsf{G}(e[\underline{a}/\underline{x}]) =_\lambda \mathsf{G}(e)[x_1 := \mathsf{G}(a_1)] \ldots [x_n := \mathsf{G}(a_n)] - \beta$$

Proof. The proof is by induction on the construction of the expression e by using 1^*, 2^*, 3^* and 4^*.

It is now possible to prove that the map G preserves equality.

Theorem A.6.12 *If* $a = b - \beta$ *then* $\mathsf{G}(a) =_\lambda \mathsf{G}(b) - \beta$

Proof. The proof is by induction on the derivation of $a = b - \beta$.

The next theorems will prove that expressions and typed lambda-calculus are indeed very similar. Let us begin with

Theorem A.6.13 *Let* $a - \alpha \ \mathsf{exp}^\lambda$ *and* $c - \gamma \ \mathsf{exp}$. *Then* $a =_\lambda \mathsf{G}(\mathsf{F}(a)) - \alpha$ *and* $c = \mathsf{F}(\mathsf{G}(c)) - \gamma$.

Proof. In both cases the proof is by induction on the construction of the considered expression.

We can now conclude

Corollary A.6.14 *The* System^λ *and* Exp *are isomorphic, that is*

1. $a =_\lambda b - \alpha$ *if and only if* $\mathsf{F}(a) = \mathsf{F}(b) - \alpha$
2. $a = b - \alpha$ *if and only if* $\mathsf{G}(a) =_\lambda \mathsf{G}(b) - \alpha$

Proof. To prove point (1), one direction is just theorem A.6.8. The other one is an immediate consequence of the previous theorems. In fact, from

$$\mathsf{F}(a) = \mathsf{F}(b) - \alpha$$

it follows

$$\begin{array}{lll} a & =_\lambda & \mathsf{G}(\mathsf{F}(a)) \\ & =_\lambda & \mathsf{G}(\mathsf{F}(b)) \\ & =_\lambda & b \end{array}$$

The proof of point (2) is completely analogous.

It easy to extend the above results to the case of a typed System^λ where also abbreviating definition, in the style of our expression theory, are introduced. In this case we can eliminate these definitions just by translating the λ-term that contains them into an expression. Then we can normalize the expression so obtained and hence obtain again a λ-term, equivalent to the starting one, without definitions.

Appendix B

The complete rule system

B.1 The forms of judgements

$$A \text{ type } [\Gamma]$$

$$A = C \ [\Gamma]$$

$$a \in A \ [\Gamma]$$

$$a = c \in A \ [\Gamma]$$

B.2 The structural rules

B.2.1 Weakening

Let Γ' be a context extending Γ. Then the following rule is valid

$$\frac{F \ [\Gamma]}{F \ [\Gamma']}$$

B.2.2 Assumptions rules

Let $a_1 : A_1, \ldots, a_n : A_n$ fit with $w_1 : A_1, \ldots, w_n : A_n$ and $a_1 = c_1 : A_1 \ldots a_n = c_n : A_n$ fit with $w_1 : A_1, \ldots, w_n : A_n$. Then

$$\frac{\begin{array}{c}[w_1 : A_1, \ldots, w_n : A_n] \\ \vdots \\ a_1 : A_1 \ldots a_n : A_n \qquad B(w_1, \ldots, w_n) \text{ type}\end{array}}{x(a_1, .., a_n) \in B(a_1, .., a_n) \; [x : (w_1 : A_1, .., w_n : A_n)B(w_1, .., w_n)]}$$

$$\frac{\begin{array}{c}[w_1 : A_1, \ldots, w_n : A_n] \\ \vdots \\ a_1 = c_1 : A_1 \ldots a_n = c_n : A_n \qquad B(w_1, \ldots, w_n) \text{ type}\end{array}}{x(a_1, .., a_n) = x(c_1, .., c_n) \in B(a_1, .., a_n)[x : (w_1 : A_1, .., w_n : A_n)B(w_1, .., w_n)]}$$

B.2.3 Equality rules

$$\begin{array}{lll} \text{(Reflexivity)} & \dfrac{a \in A}{a = a \in A} & \dfrac{A \text{ type}}{A = A} \\ \text{(Simmetry)} & \dfrac{a = c \in A}{c = a \in A} & \dfrac{A = C}{C = A} \\ \text{(Transitivity)} & \dfrac{a = e \in A \quad e = c \in A}{a = c \in A} & \dfrac{A = E \quad E = C}{A = C} \end{array}$$

B.2.4 Equal types rules

$$\frac{a \in A \quad A = C}{a \in C}$$

$$\frac{a = c \in A \quad A = C}{a = c \in C}$$

B.2.5 Substitution rules

Let $e_1 : E_1, \ldots, e_n : E_n$ and $e_1 = f_1 : E_1, \ldots, e_n = f_n : E_n$ fit with $y_1 : E_1, \ldots, y_n : E_n$. Then

$$\frac{\begin{array}{c}[y_1 : E_1, \ldots, y_n : E_n] \\ \vdots \\ e_1 : E_1 \ldots e_n : E_n \qquad A(y_1, \ldots, y_n)\ \mathsf{type}\end{array}}{A(e_1, \ldots, e_n)\ \mathsf{type}}$$

$$\frac{\begin{array}{c}[y_1 : E_1, \ldots, y_n : E_n] \\ \vdots \\ e_1 = f_1 : E_1 \ldots e_n = f_n : E_n \qquad A(y_1, \ldots, y_n)\ \mathsf{type}\end{array}}{A(e_1, \ldots, e_n) = A(f_1, \ldots, f_n)}$$

$$\frac{\begin{array}{c}[y_1 : E_1, \ldots, y_n : E_n] \\ \vdots \\ e_1 : E_1 \ldots e_n : E_n \quad A(y_1, \ldots, y_n) = C(y_1, \ldots, y_n)\end{array}}{A(e_1, \ldots, e_n) = C(e_1, \ldots, e_n)}$$

$$\frac{\begin{array}{c}[y_1 : E_1, \ldots, y_n : E_n] \\ \vdots \\ e_1 : E_1 \ldots e_n : E_n \quad a(y_1, \ldots, y_n) \in A(y_1, \ldots, y_n)\end{array}}{a(e_1, \ldots, e_n) \in A(e_1, \ldots, e_n)}$$

$$\frac{\begin{array}{c}[y_1 : E_1, .., y_n : E_n] \\ \vdots \\ e_1 = f_1 : E_1 \ldots e_n = f_n : E_n \quad a(y_1, .., y_n) \in A(y_1, .., y_n)\end{array}}{a(e_1, \ldots, e_n) = a(f_1, \ldots, f_n) \in A(e_1, \ldots, e_n)}$$

$$\frac{\begin{array}{c}[y_1 : E_1, .., y_n : E_n] \\ \vdots \\ e_1 : E_1 \ldots e_n : E_n \quad a(y_1, .., y_n) = c(y_1, .., y_n) \in A(y_1, .., y_n)\end{array}}{a(e_1, \ldots, e_n) = c(e_1, \ldots, e_n) \in A(e_1, \ldots, e_n)}$$

B.3 The logical rules

B.3.1 Π-rules

In the following rules we suppose that $x - 0$ var, $y - (0)0$ var and $t - 0$ var.

Formation

$$\frac{\begin{array}{c}[x:A]\\ \vdots\\ A \text{ type} \quad B(x) \text{ type}\end{array}}{\Pi(A,B) \text{ type}} \qquad \frac{\begin{array}{c}[x:A]\\ \vdots\\ A = C \quad B(x) = D(x)\end{array}}{\Pi(A,B) = \Pi(C,D)}$$

Introduction

$$\frac{\begin{array}{c}[x:A]\\ \vdots\\ b(x) \in B(x) \quad A \text{ type} \quad B(x) \text{ type}\end{array}}{\lambda(b) \in \Pi(A,B)}$$

$$\frac{\begin{array}{c}[x:A]\\ \vdots\\ b(x) = d(x) \in B(x) \quad A \text{ type} \quad B(x) \text{ type}\end{array}}{\lambda(b) = \lambda(d) \in \Pi(A,B)}$$

Elimination

$$\frac{c \in \Pi(A,B) \quad \begin{array}{c}[y:(x:A)\ B(x)]\\ \vdots\\ d(y) \in C(\lambda(y))\end{array} \quad \begin{array}{c}[t:\Pi(A,B)]\\ \vdots\\ C(t) \text{ type}\end{array}}{\mathsf{F}(c,d) \in C(c)}$$

$$\frac{c = c' \in \Pi(A,B) \quad \begin{array}{c}[y:(x:A)\ B(x)]\\ \vdots\\ d(y) = d'(y) \in C(\lambda(y))\end{array} \quad \begin{array}{c}[t:\Pi(A,B)]\\ \vdots\\ C(t) \text{ type}\end{array}}{\mathsf{F}(c,d) = \mathsf{F}(c',d') \in C(c)}$$

Equality

$$\frac{\begin{array}{c}[x:A]\\ \vdots\\ b(x) \in B(x) \quad \Pi(A,B) \text{ type}\end{array} \quad \begin{array}{c}[y:(x:A)\ B(x)]\\ \vdots\\ d(y) \in C(\lambda(y))\end{array} \quad \begin{array}{c}[t:\Pi(A,B)]\\ \vdots\\ C(t) \text{ type}\end{array}}{\mathsf{F}(\lambda(b),d) = d(b) \in C(\lambda(b))}$$

Computation

$$\Pi(A,B) \Rrightarrow \Pi(A,B)$$

$$\lambda(b) \Rightarrow \lambda(b) \qquad \frac{c \Rightarrow \lambda(b) \quad d(b) \Rightarrow g}{\mathsf{F}(c,d) \Rightarrow g}$$

B.3.2 Σ-rules

In the following rules $x - 0$ var, $y - 0$ var, $t - 0$ var.

Formation

$$\frac{\begin{array}{c}[x:A]\\ \vdots\\ A\ \mathsf{type} \quad B(x)\ \mathsf{type}\end{array}}{\Sigma(A,B)\ \mathsf{type}} \qquad \frac{\begin{array}{c}[x:A]\\ \vdots\\ A = C \quad B(x) = D(x)\end{array}}{\Sigma(A,B) = \Sigma(C,D)}$$

Introduction

$$\frac{\begin{array}{c}[x:A]\\ \vdots\\ a \in A \quad b \in B(a) \quad A\ \mathsf{type} \quad B(x)\ \mathsf{type}\end{array}}{\langle a,b\rangle \in \Sigma(A,B)}$$

$$\frac{\begin{array}{c}[x:A]\\ \vdots\\ a = c \in A \quad b = d \in B(a) \quad A\ \mathsf{type} \quad B(x)\ \mathsf{type}\end{array}}{\langle a,b\rangle = \langle c,d\rangle \in \Sigma(A,B)}$$

Elimination

$$\frac{\begin{array}{ccc}[x:A, y:B(x)] & & [t:\Sigma(A,B)]\\ \vdots & & \vdots\\ c \in \Sigma(A,B) \quad d(x,y) \in C(\langle x,y\rangle) & & C(t)\ \mathsf{type}\end{array}}{\mathsf{E}(c,d) \in C(c)}$$

$$\frac{\begin{array}{ccc}[x:A, y:B(x)] & & [t:\Sigma(A,B)]\\ \vdots & & \vdots\\ c = c' \in \Sigma(A,B) \quad d(x,y) = d'(x,y) \in C(\langle x,y\rangle) & & C(t)\ \mathsf{type}\end{array}}{\mathsf{E}(c,d) = \mathsf{E}(c',d') \in C(c)}$$

Equality

$$\frac{\begin{array}{ccc}[x:A, y:B(x)] & & [t:S(A,B)]\\ \vdots & & \vdots\\ a \in A \quad b \in B(a) \quad \Sigma(A,B)\mathsf{type} \quad d(x,y) \in C(\langle x,y\rangle) & & C(t)\ \mathsf{type}\end{array}}{\mathsf{E}(\langle a,b\rangle,d) = d(a,b) \in C(\langle a,b\rangle)}$$

Computation

$$\Sigma(A,B) \Rrightarrow \Sigma(A,B)$$

$$\langle a,b\rangle \Rightarrow \langle a,b\rangle \qquad \frac{c \Rightarrow \langle a,b\rangle \quad d(a,b) \Rightarrow g}{\mathsf{E}(c,d) \Rightarrow g}$$

B.3.3 +-rules

In the following rules $x - 0$ var, $y - 0$ var, $t - 0$ var.

Formation

$$\frac{A \text{ type} \quad B \text{ type}}{+(A,B) \text{ type}} \qquad \frac{A = C \quad B = D}{+(A,B) = +(C,D)}$$

Introduction

$$\frac{a \in A \quad A \text{ type} \quad B \text{ type}}{\mathsf{inl}(a) \in +(A,B)} \qquad \frac{b \in B \quad A \text{ type} \quad B \text{ type}}{\mathsf{inr}(b) \in +(A,B)}$$

$$\frac{a = c \in A \quad A \text{ type} \quad B \text{ type}}{\mathsf{inl}(a) = \mathsf{inl}(c) \in +(A,B)} \qquad \frac{b = d \in B \quad A \text{ type} \quad B \text{ type}}{\mathsf{inr}(b) = \mathsf{inr}(d) \in +(A,B)}$$

Elimination

$$\frac{c \in +(A,B) \quad \begin{array}{c}[x:A]\\ \vdots \\ d(x) \in C(\mathsf{inl}(x))\end{array} \quad \begin{array}{c}[y:B]\\ \vdots \\ e(y) \in C(\mathsf{inr}(y))\end{array} \quad \begin{array}{c}[t:+(A,B)]\\ \vdots \\ C(t) \text{ type}\end{array}}{\mathsf{D}(c,d,e) \in C(c)}$$

$$\frac{c = c' \in +(A,B) \;\; d(x) = d'(x) \in C(\mathsf{inl}(x)) \;\; e(y) = e'(y) \in C(\mathsf{inr}(y)) \;\; \begin{array}{c}[t:+(A,B)]\\ \vdots \\ C(t) \text{ type}\end{array}}{\mathsf{D}(c,d,e) = \mathsf{D}(c',d',e') \in C(c)}$$

Equality

$$\frac{a \in A \quad +(A,B) \text{ type} \quad \begin{array}{c}[x:A]\\ \vdots \\ d(x) \in C(\mathsf{inl}(x))\end{array} \quad \begin{array}{c}[y:B]\\ \vdots \\ e(y) \in C(\mathsf{inr}(y))\end{array} \quad \begin{array}{c}[t:+(A,B)]\\ \vdots \\ C(t) \text{ type}\end{array}}{\mathsf{D}(\mathsf{inl}(a),d,e) = d(a) \in C(\mathsf{inl}(a))}$$

$$\frac{b \in B \quad +(A,B) \text{ type} \quad \begin{array}{c}[x:A]\\ \vdots \\ d(x) \in C(\mathsf{inl}(x))\end{array} \quad \begin{array}{c}[y:B]\\ \vdots \\ e(y) \in C(\mathsf{inr}(y))\end{array} \quad \begin{array}{c}[t:+(A,B)]\\ \vdots \\ C(t) \text{ type}\end{array}}{\mathsf{D}(\mathsf{inr}(b),d,e) = e(b) \in C(\mathsf{inr}(b))}$$

Computation

$$+(A,B) \Rrightarrow +(A,B)$$

$$\mathsf{inl}(a) \Rightarrow \mathsf{inl}(a) \qquad \frac{c \Rightarrow \mathsf{inl}(a) \quad d(a) \Rightarrow g}{\mathsf{D}(c,d,e) \Rightarrow g}$$

$$\mathsf{inr}(b) \Rightarrow \mathsf{inr}(b) \qquad \frac{c \Rightarrow \mathsf{inr}(b) \quad e(b) \Rightarrow g}{\mathsf{D}(c,d,e) \Rightarrow g}$$

B.3.4 Eq-rules

Formation

$$\frac{A\ \mathsf{type} \quad a \in A \quad b \in A}{\mathsf{Eq}(A,a,b)\ \mathsf{type}} \qquad \frac{A = C \quad a = c \in A \quad b = d \in A}{\mathsf{Eq}(A,a,b) = \mathsf{Eq}(C,c,d)}$$

Introduction

$$\frac{a = b \in A}{\mathsf{e} \in \mathsf{Eq}(A,a,b)} \qquad \frac{a = b \in A}{\mathsf{e} = \mathsf{e} \in \mathsf{Eq}(A,a,b)}$$

Elimination

$$\frac{c \in \mathsf{Eq}(A,a,b) \quad A\ \mathsf{type} \quad a \in A \quad b \in A}{a = b \in A}$$

Equality

$$\frac{a = b \in A \quad c \in \mathsf{Eq}(A,a,b)}{c = \mathsf{e} \in \mathsf{Eq}(A,a,b)}$$

Computation

$$\mathsf{Eq}(A,a,b) \Rrightarrow \mathsf{Eq}(A,a,b)$$

$$\mathsf{e} \Rightarrow \mathsf{e}$$

B.3.5 Id-rules

In the following rules $x - 0$ var, $y - 0$ var and $z - 0$ var.

Formation

$$\frac{A \text{ type} \quad a \in A \quad b \in A}{\mathsf{Id}(A,a,b) \text{ type}} \qquad \frac{A = C \quad a = c \in A \quad b = d \in A}{\mathsf{Id}(A,a,b) = \mathsf{Id}(C,c,d)}$$

Introduction

$$\frac{a \in A}{\mathsf{r}(a) \in \mathsf{Id}(A,a,a)} \qquad \frac{a = c \in A}{\mathsf{r}(a) = \mathsf{r}(c) \in \mathsf{Id}(A,a,a)}$$

Elimination

$$\frac{\begin{array}{cc} & [x:A] \qquad\qquad [x:A, y:A, z:\mathsf{Id}(A,x,y)] \\ & \vdots \qquad\qquad\qquad\quad \vdots \\ c \in \mathsf{Id}(A,a,b) \quad d(x) \in C(x,x,\mathsf{r}(x)) & \qquad\qquad C(x,y,z) \text{ type} \end{array}}{\mathsf{K}(c,d) \in C(a,b,c)}$$

$$\frac{\begin{array}{cc} & [x:A] \qquad\qquad [x:A, y:A, z:\mathsf{Id}(A,x,y)] \\ & \vdots \qquad\qquad\qquad\quad \vdots \\ c = c' \in \mathsf{Id}(A,a,b) \quad d(x) = d'(x) \in C(x,x,\mathsf{r}(x)) & \qquad\qquad C(x,y,z) \text{ type} \end{array}}{\mathsf{K}(c,d) = \mathsf{K}(c',d') \in C(a,b,c)}$$

Equality

$$\frac{\begin{array}{cc} & [x:A] \qquad\qquad [x:A, y:A, z:\mathsf{Id}(A,x,y)] \\ & \vdots \qquad\qquad\qquad\quad \vdots \\ a \in A \quad d(x) \in C(x,x,\mathsf{r}(x)) & \qquad\qquad C(x,y,z) \text{ type} \end{array}}{\mathsf{K}(\mathsf{r}(a),d) = d(a) \in C(a,a,\mathsf{r}(a))}$$

Computation

$$\mathsf{Id}(A,a,b) \Rightarrow \mathsf{Id}(A,a,b)$$

$$\mathsf{r}(a) \Rightarrow \mathsf{r}(a) \qquad\qquad \frac{c \Rightarrow \mathsf{r}(a) \quad d(a) \Rightarrow g}{\mathsf{K}(c,d) \Rightarrow g}$$

B.3.6 $\mathsf{S}(A)$-rules

In the following rules $x - 0$ var and $t - 0$ var.

Formation

$$\frac{A\mathsf{type}}{\mathsf{S}(A)\ \mathsf{type}} \qquad \frac{A = B}{\mathsf{S}(A) = \mathsf{S}(B)}$$

Introduction

$$0_{\mathsf{S}(A)} \in \mathsf{S}(A) \qquad \frac{a \in A}{\mathsf{s}_{\mathsf{S}(A)}(a) \in \mathsf{S}(A)}$$

$$0_{\mathsf{S}(A)} = 0_{\mathsf{S}(A)} \in \mathsf{S}(A) \qquad \frac{a = b \in A}{\mathsf{s}_{\mathsf{S}(A)}(a) = \mathsf{s}_{\mathsf{S}(A)}(b) \in \mathsf{S}(A)}$$

Elimination

$$\frac{c \in \mathsf{S}(A) \quad d \in C(0_{\mathsf{S}(A)}) \quad e(x) \in C(\mathsf{s}_{\mathsf{S}(A)}(x)) \quad C(t)\ \mathsf{type}}{\mathsf{S}_{rec}(c, d, e) \in C(c)}$$

where $e(x) \in C(\mathsf{s}_{\mathsf{S}(A)}(x))$ is derived from $[x : A]$ and $C(t)\ \mathsf{type}$ is derived from $[t : \mathsf{S}(A)]$.

$$\frac{c = c' \in \mathsf{S}(A) \quad d = d' \in C(0_{\mathsf{S}(A)}) \quad e(x) = e'(x) \in C(\mathsf{s}_{\mathsf{S}(A)}(x)) \quad C(t)\ \mathsf{type}}{\mathsf{S}_{rec}(c, d, e) = \mathsf{S}_{rec}(c', d', e') \in C(c)}$$

where $e(x) = e'(x) \in C(\mathsf{s}_{\mathsf{S}(A)}(x))$ is derived from $[x : A]$ and $C(t)\ \mathsf{type}$ is derived from $[t : \mathsf{S}(A)]$.

Equality

$$\frac{d \in C(0_{\mathsf{S}(A)}) \quad e(x) \in C(\mathsf{s}_{\mathsf{S}(A)}(x)) \quad C(t)\ \mathsf{type}}{\mathsf{S}_{rec}(0_{\mathsf{S}(A)}, d, e) = d \in C(0_{\mathsf{S}(A)})}$$

where $e(x) \in C(\mathsf{s}_{\mathsf{S}(A)}(x))$ is derived from $[x : A]$ and $C(t)\ \mathsf{type}$ is derived from $[t : \mathsf{S}(A)]$.

$$\frac{a \in A \quad d \in C(0_{\mathsf{S}(A)}) \quad e(x) \in C(\mathsf{s}_{\mathsf{S}(A)}(x)) \quad C(t)\ \mathsf{type}}{\mathsf{S}_{rec}(\mathsf{s}_{\mathsf{S}(A)}(a), d, e) = e(a) \in C(\mathsf{s}_{\mathsf{S}(A)}(a))}$$

where $e(x) \in C(\mathsf{s}_{\mathsf{S}(A)}(x))$ is derived from $[x : A]$ and $C(t)\ \mathsf{type}$ is derived from $[t : \mathsf{S}(A)]$.

Computation

$$\mathsf{S}(A) \Rrightarrow \mathsf{S}(A)$$

$$0_{\mathsf{S}(A)} \Rightarrow 0_{\mathsf{S}(A)} \qquad \frac{c \Rightarrow 0_{\mathsf{S}(A)} \quad d \Rightarrow g}{\mathsf{S}_{rec}(c, d, e) \Rightarrow g}$$

$$\mathsf{s}_{\mathsf{S}(A)}(a) \Rightarrow \mathsf{s}_{\mathsf{S}(A)}(a) \qquad \frac{c \Rightarrow \mathsf{s}_{\mathsf{S}(A)}(a) \quad e(a) \Rightarrow g}{\mathsf{S}_{rec}(c, d, e) \Rightarrow g}$$

B.3.7 N_n-rules

In the following rules $t - 0$ var.

Formation

$$\mathsf{N}_n \text{ type} \qquad \mathsf{N}_n = \mathsf{N}_n$$

Introduction

$$0_n \in \mathsf{N}_n, \ldots, \mathsf{m}_n \in \mathsf{N}_n, \ldots, \mathsf{n}-1_n \in \mathsf{N}_n$$

$$0_n = 0_n \in \mathsf{N}_n, \ldots, \mathsf{m}_n = \mathsf{m}_n \in \mathsf{N}_n, \ldots, \mathsf{n}-1_n = \mathsf{n}-1_n \in \mathsf{N}_n$$

Elimination

$$\frac{\begin{array}{c}[t : \mathsf{N}_n] \\ \vdots \\ c \in \mathsf{N}_n \quad d_0 \in C(0_n) \ldots d_{n-1} \in C(\mathsf{n}-1_n) \quad C(t) \text{ type}\end{array}}{\mathsf{Rec}_n(c, d_0, \ldots, d_{n-1}) \in C(c)}$$

$$\frac{\begin{array}{c}[t : \mathsf{N}_n] \\ \vdots \\ c = c' \in \mathsf{N}_n \quad d_0 = d'_0 \in C(0_n) \ldots d_{n-1} = d'_{n-1} \in C(\mathsf{n}-1_1) \quad C(t) \text{ type}\end{array}}{\mathsf{Rec}_n(c, d_0, \ldots, d_{n-1}) = \mathsf{Rec}_n(c', d'_0, \ldots, d'_{n-1}) \in C(c)}$$

Note that if $n = 0$ this is the usual $\bot$-rule.

Equality

$$\frac{\begin{array}{c}[t : \mathsf{N}_n] \\ \vdots \\ d_0 \in C(0_n) \ldots d_{n-1} \in C(\mathsf{n}-1_n) \quad C(t) \text{ type}\end{array}}{\mathsf{Rec}_n(\mathsf{m}_n, d_0, \ldots, d_{n-1}) = d_m \in C(\mathsf{m}_n)}$$

Note that N_n has n equality rules and hence N_0 has no equality rule.

Computation

$$\mathsf{N}_n \Rrightarrow \mathsf{N}_n$$

$$\mathsf{m}_n \Rightarrow \mathsf{m}_n \quad \frac{c \Rightarrow \mathsf{m}_n \quad d_m \Rightarrow g}{\mathsf{Rec}_n(c, d_0, \ldots, d_{n-1}) \Rightarrow g}$$

Note that N_n has n computation rules for canonical elements and n computation rules for non canonical elements. In particular, N_0 has no computation rule.

B.3.8 N-rules

In the following rules $x - 0$ var, $y - 0$ var, $t - 0$ var.

Formation

$$\mathsf{N}\ \mathsf{type} \qquad \mathsf{N} = \mathsf{N}$$

Introduction

$$0 \in \mathsf{N} \qquad \frac{a \in \mathsf{N}}{\mathsf{s}(a) \in \mathsf{N}}$$

$$0 = 0 \in \mathsf{N} \qquad \frac{a = b \in \mathsf{N}}{\mathsf{s}(a) = \mathsf{s}(b) \in \mathsf{N}}$$

Elimination

$$\frac{c \in \mathsf{N} \quad d \in C(0) \quad \begin{array}{c}[x : \mathsf{N}, y : C(x)] \\ \vdots \\ e(x,y) \in C(\mathsf{s}(x))\end{array} \quad \begin{array}{c}[t : \mathsf{N}] \\ \vdots \\ C(t)\ \mathsf{type}\end{array}}{\mathsf{N}_{rec}(c,d,e) \in C(c)}$$

$$\frac{c = c' \in \mathsf{N} \quad d = d' \in C(0) \quad \begin{array}{c}[x : \mathsf{N}, y : C(x)] \\ \vdots \\ e(x,y) = e'(x,y) \in C(\mathsf{s}(x))\end{array} \quad \begin{array}{c}[t : \mathsf{N}] \\ \vdots \\ C(t)\ \mathsf{type}\end{array}}{\mathsf{N}_{rec}(c,d,e) = \mathsf{N}_{rec}(c',d',e') \in C(c)}$$

Equality

$$\frac{d \in C(0) \quad \begin{array}{c}[x : \mathsf{N}, y : C(x)] \\ \vdots \\ e(x,y) \in C(\mathsf{s}(x))\end{array} \quad \begin{array}{c}[t : \mathsf{N}] \\ \vdots \\ C(t)\ \mathsf{type}\end{array}}{\mathsf{N}_{rec}(0,d,e) = d \in C(0)}$$

$$\frac{c \in N \quad d \in C(0) \quad \begin{array}{c}[x : \mathsf{N}, y : C(x)] \\ \vdots \\ e(x,y) \in C(\mathsf{s}(x))\end{array} \quad \begin{array}{c}[t : \mathsf{N}] \\ \vdots \\ C(t)\ \mathsf{type}\end{array}}{\mathsf{N}_{rec}(\mathsf{s}(c),d,e) = e(c,\mathsf{N}_{rec}(c,d,e)) \in C(\mathsf{s}(c))}$$

Computation

$$\mathsf{N} \Rrightarrow \mathsf{N}$$

$$0 \Rightarrow 0 \qquad \frac{c \Rightarrow 0 \quad d \Rightarrow g}{\mathsf{N}_{rec}(c,d,e) \Rightarrow g}$$

$$\mathsf{s}(a) \Rightarrow \mathsf{s}(a) \qquad \frac{c \Rightarrow \mathsf{s}(a) \quad e(a,\mathsf{N}_{rec}(a,d,e)) \Rightarrow g}{\mathsf{N}_{rec}(c,d,e) \Rightarrow g}$$

B.3.9 W-rules

In the following rules $x - 0$ var, $u - 0$ var, $y - (0)0$ var, $z - (0)0$ var and $t - 0$ var.

Formation

$$\frac{\begin{array}{c}[x:A]\\ \vdots\\ A \text{ type} \quad B(x) \text{ type}\end{array}}{\mathsf{W}(A,B) \text{ type}} \qquad \frac{\begin{array}{c}[x:A]\\ \vdots\\ A = C \quad B(x) = D(x)\end{array}}{\mathsf{W}(A,B) = \mathsf{W}(C,D)}$$

Introduction

$$\frac{\begin{array}{c}[x:B(a)]\\ \vdots\\ a \in A \quad b(x) \in \mathsf{W}(A,B) \quad A \text{ type} \quad B(x) \text{ type}\end{array}}{\mathsf{sup}(a,b) \in \mathsf{W}(A,B)}$$

$$\frac{\begin{array}{c}[x:B(a)]\\ \vdots\\ a = c \in A \quad b(x) = d(x) \in \mathsf{W}(A,B) \quad A \text{ type} \quad B(x) \text{ type}\end{array}}{\mathsf{sup}(a,b) = \mathsf{sup}(c,d) \in \mathsf{W}(A,B)}$$

Elimination

$$\frac{\begin{array}{ccc}[x:A, y:(t:B(x))\mathsf{W}(A,B), z:(u:B(x))C(y(u))] & & [t:\mathsf{W}(A,B)]\\ \vdots & & \vdots\\ c \in \mathsf{W}(A,B) \qquad d(x,y,z) \in C(\mathsf{sup}(x,y)) & & C(t) \text{ type}\end{array}}{\mathsf{T}_{rec}(c,d) \in C(c)}$$

$$\frac{\begin{array}{ccc}[x:A, y:(t:B(x))\mathsf{W}(A,B), z:(u:B(x))C(y(u))] & & [t:\mathsf{W}(A,B)]\\ \vdots & & \vdots\\ c = c' \in \mathsf{W}(A,B) \quad d(x,y,z) = d'(x,y,z) \in C(\mathsf{sup}(x,y)) & & C(t) \text{ type}\end{array}}{\mathsf{T}_{rec}(c,d) = \mathsf{T}_{rec}(c',d') \in C(c)}$$

Equality

$$\frac{\begin{array}{ccccc} & & [x:A, y:(t:B(x))\mathsf{W}(A,B), z:(u:B(x))C(y(u))] & & \\ & [x:B(a)] & \vdots & & [t:\mathsf{W}(A,B)]\\ a \in A \quad b(x) \in \mathsf{W}(A,B) & & d(x,y,z) \in C(\mathsf{sup}(x,y)) & & C(t) \text{ type}\end{array}}{\mathsf{T}_{rec}(\mathsf{sup}(a,b),d) = d(a,b,(x)\ \mathsf{T}_{rec}(b(x),d)) \in C(\mathsf{sup}(a,b))}$$

Computation

$$\mathsf{W}(A,B) \Rightarrow \mathsf{W}(A,B)$$

$$\mathsf{sup}(a,b) \Rightarrow \mathsf{sup}(a,b) \qquad \frac{c \Rightarrow \mathsf{sup}(a,b) \quad d(a,b,(x)\ \mathsf{T}_{rec}(b(x),d)) \Rightarrow g}{\mathsf{T}_{rec}(c,d) \Rightarrow g}$$

B.3.10 U-rules

In the following rules $x - 0$ var.

Formation

$$\mathsf{U}\ \mathsf{type} \qquad \mathsf{U} = \mathsf{U}$$

Introduction

$$\frac{\begin{array}{c}[x :< a >] \\ \vdots \\ a \in \mathsf{U} \quad b(x) \in \mathsf{U}\end{array}}{\pi(a,b) \in \mathsf{U}} \qquad \frac{\begin{array}{c}[x :< a >] \\ \vdots \\ a = c \in \mathsf{U} \quad b(x) = d(x) \in \mathsf{U}\end{array}}{\pi(a,b) = \pi(c,d) \in \mathsf{U}}$$

$$\frac{\begin{array}{c}[x :< a >] \\ \vdots \\ a \in \mathsf{U} \quad b(x) \in \mathsf{U}\end{array}}{\sigma(a,b) \in \mathsf{U}} \qquad \frac{\begin{array}{c}[x :< a >] \\ \vdots \\ a = c \in \mathsf{U} \quad b(x) = d(x) \in \mathsf{U}\end{array}}{\sigma(a,b) = \sigma(c,d) \in \mathsf{U}}$$

$$\frac{a \in \mathsf{U} \quad b \in \mathsf{U}}{+(a,b) \in \mathsf{U}} \qquad \frac{a = c \in \mathsf{U} \quad b = d \in \mathsf{U}}{+(a,b) = +(c,d) \in \mathsf{U}}$$

$$\frac{a \in \mathsf{U} \quad b \in\!< a > \quad d \in\!< a >}{\mathsf{eq}(a,b,d) \in \mathsf{U}} \qquad \frac{a = c \in \mathsf{U} \quad b = e \in\!< a > \quad d = f \in\!< a >}{\mathsf{eq}(a,b,d) = \mathsf{eq}(c,e,f) \in \mathsf{U}}$$

$$\frac{a \in \mathsf{U} \quad b \in\!< a > \quad d \in\!< a >}{\mathsf{id}(a,b,d) \in \mathsf{U}} \qquad \frac{a = c \in \mathsf{U} \quad b = e \in\!< a > \quad d = f \in\!< a >}{\mathsf{id}(a,b,d) = \mathsf{id}(c,e,f) \in \mathsf{U}}$$

$$\frac{a \in \mathsf{U}}{\mathsf{s}(a) \in \mathsf{U}} \qquad \frac{a = c \in \mathsf{U}}{\mathsf{s}(a) = \mathsf{s}(c) \in \mathsf{U}}$$

$$\mathsf{n}_n \in \mathsf{U} \qquad \mathsf{n}_n = \mathsf{n}_n \in \mathsf{U}$$

$$\mathsf{n} \in \mathsf{U} \qquad \mathsf{n} = \mathsf{n} \in \mathsf{U}$$

$$\frac{\begin{array}{c}[x :< a >] \\ \vdots \\ a \in \mathsf{U} \quad b(x) \in \mathsf{U}\end{array}}{\mathsf{w}(a,b) \in \mathsf{U}} \qquad \frac{\begin{array}{c}[x :< a >] \\ \vdots \\ a = c \in \mathsf{U} \quad b(x) = d(x) \in \mathsf{U}\end{array}}{\mathsf{w}(a,b) = \mathsf{w}(c,d) \in \mathsf{U}}$$

Elimination

$$\frac{a \in \mathsf{U}}{< a > \ \mathsf{type}} \qquad \frac{a = b \in \mathsf{U}}{< a >=< b >}$$

Equality

$$\frac{\begin{array}{c}[x :< a >]\\ \vdots\\ a \in \mathsf{U} \quad b(x) \in \mathsf{U}\end{array}}{< \pi(a,b) >= \Pi(< a >, (x) < b(x) >)} \qquad \frac{\begin{array}{c}[x :< a >]\\ \vdots\\ a \in \mathsf{U} \quad b(x) \in \mathsf{U}]\end{array}}{< \sigma(a,b) >= \Sigma(< a >, (x) < b(x) >)}$$

$$\frac{a \in \mathsf{U} \quad b \in \mathsf{U}}{< +(a,b) >= +(< a >, < b >)} \qquad \frac{\begin{array}{c}[x :< a >]\\ \vdots\\ a \in \mathsf{U} \quad b(x) \in \mathsf{U}\end{array}}{< \mathsf{w}(a,b) >= \mathsf{W}(< a >, (x) < b(x) >)}$$

$$\frac{a \in \mathsf{U} \quad b \in< a > \quad d \in< a >}{< \mathsf{eq}(a,b,d) >= \mathsf{Eq}(< a >, b, d)} \qquad \frac{a \in \mathsf{U} \quad b \in< a > \quad d \in< a >}{< \mathsf{id}(a,b,d) >= \mathsf{Id}(< a >, b, d)}$$

$$\frac{a \in \mathsf{U}}{< \mathsf{s}(a) >= \mathsf{S}(< a >)}$$

$$< \mathsf{n}_n >= \mathsf{N}_n \qquad < \mathsf{n} >= \mathsf{N}$$

Computation

$$\mathsf{U} \Rrightarrow \mathsf{U}$$

$$\pi(a,b) \Rightarrow \pi(a,b) \qquad \frac{c \Rightarrow \pi(a,b)}{< c >\Rrightarrow \Pi(< a >, (x) \ < b(x) >)}$$

$$\sigma(a,b) \Rightarrow \sigma(a,b) \qquad \frac{c \Rightarrow \sigma(a,b)}{< c >\Rrightarrow \Sigma(< a >, (x) \ < b(x) >)}$$

$$+(a,b) \Rightarrow +(a,b) \qquad \frac{c \Rightarrow +(a,b)}{< c >\Rrightarrow +(< a >, < b >)}$$

$$\mathsf{eq}(a,b,d) \Rightarrow \mathsf{eq}(a,b,d) \qquad \frac{c \Rightarrow \mathsf{eq}(a,b,d)}{< c >\Rrightarrow \mathsf{Eq}(< a >, b, d)}$$

$$\mathsf{id}(a,b,d) \Rightarrow \mathsf{id}(a,b,d) \qquad \frac{c \Rightarrow \mathsf{id}(a,b,d)}{< c >\Rrightarrow \mathsf{Id}(< a >, b, d)}$$

$$\mathsf{s}(a) \Rightarrow \mathsf{s}(a) \qquad \frac{c \Rightarrow \mathsf{s}(a)}{< c >\Rrightarrow \mathsf{S}(< a >)}$$

$$\mathsf{n}_n \Rightarrow \mathsf{n}_n \qquad \frac{c \Rightarrow \mathsf{n}_n}{< c >\Rrightarrow \mathsf{N}_n}$$

$$\mathsf{n} \Rightarrow \mathsf{n} \qquad \frac{c \Rightarrow \mathsf{n}}{< c >\Rrightarrow \mathsf{N}}$$

$$\mathsf{w}(a,b) \Rightarrow \mathsf{w}(a,b) \qquad \frac{c \Rightarrow \mathsf{w}(a,b)}{< c >\Rrightarrow \mathsf{W}(< a >, (x) \ < b(x) >)}$$

Bibliography

[Acz78] Aczel, P., *The type theoretic interpretation of constructive set theory*, in "Logic Colloquium '77", MacIntyre, A., Pacholski, L., Paris, J. eds., North Holland, Amsterdam, 1978.

[Acz82] Aczel, P., *The type theoretic interpretation of constructive set theory: choice principles*, in "The L.E.J. Brouwer Centenary Symposium", Troelstra, S.S., van Dalen, D. eds., North Holland, Amsterdam, 1982.

[Acz86] Aczel, P., *The type theoretic interpretation of constructive set theory; inductive definitions*, in "Logic, Metodology and Philosophy of Science VII", Marcus, R.B. et al. (eds.), North Holland, Amsterdam, 1986.

[All86] Allen, S.F., *A non-type-theoretic definition of Martin-Löf's types*, Proceedings of "The 1st Annual Symposium on Logic in Computer Science", IEEE, 1986, pp. 215-221.

[All87] Allen, S.F., *A non-type-theoretic semantics for a type-theoretic language*, PhD Thesis, Cornell University, 1987.

[BCMS89] Backhouse, R., Chisholm, P., Malcom, G. and Saaman, E., *Do-it-yourself type theory*, Formal Aspects of Computing 1, 1989, pp.19-84.

[BC85] Bates, J.L. and Constable, R.L., *Proofs as Programs*, ACM Transactions on Programming Languages and Systems 7, N.1, 1985, pp. 94-117.

[Bar84] Barendregt, H.P., *The Lambda Calculus, its Syntax and Semantics*, North–Holland, Amsterdam, 1984.

[Bar92] Barendregt, H.P., *Lambda-calculi with types*, in "Handbook of logic and computer science", vol. 2, S. Abramski, D.M. Gabbay and T.S. Maibaum eds., Oxford University Press, 1992, pp. 118-309.

[Bee85] Beeson, M.J., *Foundation of Constructive Mathematics*, Lecture Notes in Computer Science, Springer-Verlag, 1985.

[Bel88] Bell, J.L., *Toposes and local set theory: an introduction*. Clarendon Press, Oxford, 1988.

[Ber90] Berardi, S., *Type dependence and constructive mathematics*, PhD thesis, Dipartimento di Matematica, Università di Torino, 1990.

[BV87] Bossi, A. and Valentini, S., *Assunzioni di arietà superiore nella teoria intuizionistica dei tipi*, in Italian, in "Atti secondo convegno nazionale sulla Programmazione Logica", Torino 1987, B. Demo ed., pp. 81-92.

[BV89] Bossi, A. and Valentini, S., *The expressions with arity*, internal report of "Dipartimento di Scienze dell'Informazione", University of Milan, 1989, 61/89.

[BV92] Bossi, A. and Valentini, S., *An intuitionistic theory of types with assumptions of high-arity variables*, Annals of Pure and Applied Logic 57, North Holland, 1992, pp. 93-149.

[Coq90] Coquand, T., *Metamathematical investigations of a calculus of constructions*, in "Logic and Computer Science", P. Odifreddi ed., Academic Press, London, 1990, pp. 91-122.

[Coq96] Coquand, T., *An Algorithm for Type-Checking Dependent Types*, Science of Computer Programming 26 (1–3), 1996, pp. 167-177.

[CCo98] Coquand, C., *A realizability interpretation of Martin-Löf's type theory*, In "Twenty five years of Constructive Type Theory", J. Smith and G. Sambin eds., Oxford Logic Guides (36), Clarendon Press, Oxford, 1998, pp. 73-82.

[CSSV] Coquand, T., G. Sambin, J. Smith and S. Valentini, *Inductive generation of formal topologies*, to appear.

[CV98] Capretta, V. and Valentini, S., *A general method to prove the normalization theorem for first and second order typed λ-calculi* Mathematical Structures in Computer Science, 1998, pp. 719-739.

[Chu36] Church, A. *An unsolvable problem of elementary number theory*, American Journal of Mathematics, 58 (1936), pp. 345-363.

[CF74] Curry, H.B., Feys, R., *Combinatory Logic*, North-Holland, Amsterdam, 1974.

[Bru80] de Bruijn, N.G., *A survey of the project Automath*, in "To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism", J.P. Seldin and J.R. Hyndley eds., Academic Press, London, 1980, pp. 589-606.

[Bru91] de Bruijn, N.G., *Telescopic mappings in typed lambda calculus*, Information and Computation, 91(2), April 1991, pp. 189-204.

[Dia75] Diaconescu, R., *Axiom of choice and complementation*, Proc. American Mathematical Society, 51, 1975, pp. 176-178.

[Fre1892] Frege, G., *Über Sinn und Bedeutung*, Zeitschrift für Philosophie und philosophische Kritik, 1892, pp. 25-50.

[Gal90] Gallier, J.H., *On Girard's "Candidats de Reductibilité"*, in "Logic and Computer Science", P. Odifreddi ed., Academic Press, London, 1990, pp. 123-203.

[Gan80] Gandy, R.O., *An early proof of normalization by A. M. Turing*, in "To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism", S. P. Seldin and J. R. Hindley eds., Academic Press, London, 1980, pp. 453-455.

[Geu93] Geuvers, H., *Logics and type systems*, PhD thesis, Katholieke Universiteit Nijmegen, The Netherlands, 1993.

[Gir71] Girard, J.Y., *Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types*, in "Proceedings of 2nd Scandinavian Logic Symposium", J. E. Fenstad ed., North–Holland, Amsterdam, 1971, pp. 63-92.

[Gir86] Girard, J.Y., *The system* F *of variable types, fifteen years later*, Theoretical Computer Science 45, North–Holland, Amsterdam, 1986, pp 159-192.

[GLT89] Girard, J.Y., Lafont, Y. and Taylor, P., *Proofs and Types*, Cambridge University Press, 1989.

[GR94] Griffor, E. and Rathjen, M., *The strength of some Martin-Löf's type theories*, Archive Mathematical Logic, 33, 1994, pp. 347-385.

[Hey56] Heyting, A., *Intuitionism, an introduction*, North-Holland, Amsterdam, 1956.

[Hof94] Hofmann, M., *Elimination of extensionality and quotient types in Martin-Löf type theory.* in "Proceedings of the International Workshop on Types for Proofs and Programs, Nijmegen, The Netherlands", H. Barendregt and T. Nipkow eds., Lecture Notes in Computer Science (806), Springer Verlag, Berlin and New York, 1994, pp. 166-190.

[Hof95] Hofmann, M., *Extensional concepts in intensional type theory*. PhD Thesis, University of Edinburgh, 1995.

[Hof97] Hofmann, M., *A simple model for quotient types*. in "Typed lambda calculi and applications", Lecture Notes in Computer Science, Springer Verlag, Berlin and New York, 1997, pp. 216–234.

[HS95] Hofmann, M. and Streicher, T., *The groupoid interpretation of type theory*, In "Twenty five years of Constructive Type Theory", J. Smith and G. Sambin eds., Oxford Logic Guides (36), Clarendon Press, Oxford, 1998, pp. 83-111.

[How80] Howard, W.A., *The formula-as-types notion of construction*, To "H.B. Curry: Essays on combinatory Logic, Lambda Calculus and Formalism" R. Hindley and J.P. Seldin eds., Academic Press, London, 1980, pp. 479-490.

[HP90] Huwig, H. and Poigné, A., *A note on inconsistencies caused by fixpoints in a cartesian closed category*, Theoretical Computer Scince, 75, 1990, pp. 101-112.

[Jac89] Jacobs, B., *The inconsistency of higher order extensions of Martin-Löf's type theory*, Journal of Philosophical Logic, 18, 1989, pp. 399-422.

[Kol32] Kolmogorov, A.N., *Zur Deutung der intuitionistischen Logik*, Mathematische Zeitschrift, vol. 35, 1932, pp. 58-65.

[Kri93] Krivine, J. L., *Lambda-Calculus, Types and Models* Masson, Paris, Ellis Horwood, Hemel Hempstead, 1993.

[LS86] Lambek, J. and Scott, P.J., *An introduction to higher order categorical logic.*, volume 7 of "Studies in Advanced Mathematics", Cambridge University Press, 1986.

[Law69] Lawvere, F.W., *Diagonal arguments and cartesian closed categories*, in "Category Theory, Homology Theory and their Applications II", Lecture Notes in Mathematics, n.92, Springer 1969, pp. 134-145.

[Luo90] Luo, Z. *An extended calculus of constructions*. PhD Thesis, University of Edinburgh, 1990.

[MV96] Maguolo, D. and Valentini, S., *An intuitionistic version of Cantor's theorem*, Mathematical Logic Quarterly 42, 1996, pp. 446-448.

[MV99] Maietti, M.E. and Valentini, S., *Can you add power-set to Martin-Löf intuitionistic set theory?* Mathematical Logic Quarterly 45, 1999, pp. 521-532.

[Mai99] Maietti, M.E., *About effective quotients in constructive type theory*, in "Types for Proofs and Programs", International Workshop "Types'98", Altenkirch T., Naraschewski W. and Reus B. eds., Lecture Notes in Computer Science 1657, Springer Verlag, 1999, pp. 164-178.

[Mag92] Magnusson, L., *The new implementation of ALF*, in proceedings of the "1992 Workshop on Types for Proofs and Programs", Nordstrom B., Petersson K. and Plotkin G. eds.

[Mat98] Matthes, R., *Extensions of* SystemF *by Iteration and Primitive Recursion on Monotone Inductive Types*, PhD thesis, Department of Mathematics and Informatics, University of Munich, 1998.

[MJ98] Matthes, R. and Joachimski, F. *Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's* T, submitted to the Archive for Mathematical Logic.

[Mar71] Martin-Löf, P. *Hauptsatz for the intuitionistic theory of iterated inductive definitions*, in "Proceedings of the second Scandinavian logic symposium", J.E. Fenstad ed., North-Holland, Amsterdam, 1971, pp. 179-216.

[Mar75] Martin-Löf, P. *An Intuitionistic Theory of Types: Predicative Part*, in "Logic Colloquium 1973", H. E. Rose and J. C. Shepherdson eds., North-Holland, Amsterdam, 1975, pp. 73-118.

[Mar82] Martin-Löf, P. *Constructive Mathematics and Computer Programming*, in "Proceedings of the 6th International Congress for Logic, Methodology and Philosophy of Science", Cohen, L.J. Los, J. Pfeiffer, H. and Podewski, K. P. eds., IV, Hannover 1979, North-Holland, Amsterdam (1982), pp.153-175.

[Mar83] Martin-Löf, P. *Siena Lectures*, handwritten notes of a series of lectures given in Siena, April 1983.

[Mar84] P. Martin-Löf, *Intuitionistic type theory*, notes by Giovanni Sambin of a series of lectures given in Padua, June 1980, Bibliopolis, Napoli, 1984.

[MNPS91] Miller, D., Nadathur, G., Pfenning, F. and Scedrov, A., *Uniform proofs as a foundation for logic programming*, Annals of Pure and Applied Logic, 51, 1991, pp. 125-157.

[MR86] Mayer, A. R. and Reinhold, M. B. *'Type' is not a type: preliminary report*, ACM, 1986, pp. 287–295.

[Nor81] Nordström, B., *Programming in Constructive Set Theory: some examples*, in proceeding of "ACM Conference on Functional Programming Languages and Computer Architecture", 1981, pp. 141-153.

[NP83] Nordstrom, B. and Petersson, K, *Types and Specifications*, in "Proceedings IFIP '83", R.E.A. Mason ed., Paris, Elsevier Science Publishers (North-Holland), Amsterdam 1983.

[NPS90] Nordstrom, B., Petersson, K. and Smith, J.M., *Programming in Martin-Löf's Type Theory, an introduction*, Oxford Univ. Press, Oxford, 1990.

[NS84] Nordstrom, B. and Smith, J.M., *Propositions, Types and Specifications of Programs in Martin-Löf's Type Theory*, BIT, 24, N.3 (October 1984), pp. 288-301.

[PV93] Paulus Venetus (1993) (alias Sambin, G. and Valentini, S.). *Propositum Cameriniense, sive etiam itinera certaminis ... (in italian)*, in "Atti degli Incontri di Logica Matematica vol. viii (XV Incontro)", G. Gerla, C. Toffalori and S. Tulipani eds., Camerino, pp. 115-143.

[PS86] Petersson, K. and Smith, J.M., *Program derivation in type theory: a partitioning problem*, Comput. Languages 11 (3/4), 1986, pp.161-172.

[Pot80] Pottinger, G., *A type assignment for strongly normalizable λ-terms*, in "To H.B. Curry, Essay on Combinatory Logic, Lambda Calculus and Formalism", Academic Press, New York, 1980, pp. 561-577.

[Pra65] Prawitz, D., *Natural Deduction: A Proof-Theoretical Study*, Almquist and Wiksell, Stockolm, 1965.

[Sam87] Sambin, G.. *Intuitionistic formal spaces - a first communication*, in "Mathematical Logic and its Applications", D. Skordev ed., Plenum, New York, 1987, pp. 187-204.

[Sam91] Sambin, G., *Per una dinamica nei fondamenti (in italian)*, in "Atti del Congresso: Nuovi problemi della logica e della filosofia della scienza", vol. II, G. Corsi and G. Sambin eds., CLUEB, Bologna, 1991, pp. 163-210.

[Sam97] Sambin, G., *Developing topology in a type theoretic setting*, to appear.

[SV93] Sambin, G. and Valentini, S., *Building up a tool-box for Martin-Löf's type theory (abstract)*, in "Computational Logic and Proof Theory. Proceedings of the Third Kurt Gödel Colloquium, KGC'93", G. Gottlob, A. Leitsch and D. Mundici eds., Lecture Notes in Computer Science, Springer, Berlin-Heidelberg-New York, 1993, pp. 69-70.

[SV98] Sambin, G. and Valentini, S., *Building up a tool-box for Martin-Löf intuitionistic type theory*, in "Twenty five years of Constructive Type Theory", J. Smith and G. Sambin eds., Oxford Logic Guides (36), Clarendon Press, Oxford, 1998, pp. 221-244.

[SVV96] Sambin, G., Valentini, S. and Virgili, P., *Constructive Domain Theory as a branch of Intuitionistic Pointfree Topology*, Theoretical Computer Science, 159, 1996, pp. 319-341.

[Tai67] Tait, W.W., *Intensional interpretation of functionals of finite type I*, Journal of Symbolic Logic, 32, 1967, pp. 198-212.

[Tro87] Troelstra, A. S., *On the Syntax of Martin Löf's Type Theories* Theoretical Computer Science, 51, 1987, pp. 1-26.

[Tur97] Turner, R., *Reading between the lines in constructive type theory*, Journal Logic Computation, 7-2, 1997, pp. 229-250.

[Val94] Valentini, S., *A note on a straightforward proof of normal form theorem for simply typed λ-calculi*, Bollettino dell'Unione Matematica Italiana, 8-A, 1994, pp. 207-213.

[Val95] Valentini, S., *On the decidability of the equality theory of simply typed lambda-calculi*, Bollettino dell'Unione Matematica Italiana, 9-A, 1995, pp. 83-93.

[Val95a] Valentini, S., *The multi-level typed λ-calculus*, in preparation.

[Val96] Valentini, S., *Decidability in Intuitionistic Type Theory is functionally decidable*, Mathematical Logic Quarterly, 42, 1996, pp. 300-304.

[Val96a] Valentini, S., *Another introduction to Martin-Löf's Type Theory*, in "Trends in Theoretical Informatics", R. Albrecht and H. Herre eds., Schriftenreihe der Österreichischen Computer Gesellscaft, Bd. 89, München, 1996.

[Val96b] Valentini, S., *The formal development of non-trivial programs in constructive type theory*, in "Proceedings of the Second International Conference on Massively Parellel Computing Systems", Ischia, maggio 1996, pp. 570-577

[Val98] Valentini, S., *The forget-restore principle: a paradigmatic example*, in "Twenty five years of Constructive Type Theory", J. Smith and G. Sambin eds., Oxford Logic Guide (36), Clarendon Press, Oxford, 1998, pp. 275-283.

[Val00] Valentini, S., *Extensionality versus constructivity*, to appear.